



(19) Europäisches Patentamt
European Patent Office
Office européen des brevets



(11) Publication number:

0 665 493 A2

(12)

EUROPEAN PATENT APPLICATION

(21) Application number: **95300024.7**

(51) Int. Cl.⁶: **G06F 9/45, G06F 9/44**

(22) Date of filing: **26.01.95**

(30) Priority: **28.01.94 US 187972**

(43) Date of publication of application:
02.08.95 Bulletin 95/31

(84) Designated Contracting States:
DE FR IT NL SE

(71) Applicant: **SUN MICROSYSTEMS, INC.**
2550 Garcia Avenue
Mountain View, CA 94043 (US)

(72) Inventor: **Gibbons, Jonathan J.**
269 Lassen Avenue
Mountain View,
California 94043 (US)
Inventor: **Day, Michael J.**
372 Mountain View Avenue
Mountain View,
California 94041 (US)
Inventor: **Goldstein, Theodore C.**
875 La Para Avenue
Palo Alto,
California 94306 (US)
Inventor: **Jordan, Michael J.**
784 Josina Avenue
Palo Alto,
California 94306 (US)

(74) Representative: **Wombwell, Francis et al**
Potts, Kerr & Co.
15, Hamilton Square
Birkenhead
Merseyside L41 6BR (GB)

(54) **A typesafe framework for dynamically extensible objects.**

(57) The present invention provides a system and process for making use of pre-existing data-structures which represent a computer program, in a way which has the advantages of shortening the time and cost required to create a new version of the computer program. The pre-existing data-structure is modified to produce a shadow-data-structure which contains only shadows of those elements or nodes of the pre-existing data-structure required to perform the tasks of the new version of the computer program. The present invention includes processes to make the data-structure of the original program shadowable; processes to use data from the original program compilation process in compiling the new version of the program, including processes to create a shadow data-structure; and processes to use the new version of the computer program along with the shadow data-structure to create the desired execution. This new version of the computer program is typically a tool for checking or observing the original program's execution in some manner. Moreover, the system and processes disclosed provide mechanisms for a software manufacturer to create type-safe versions of a connected collection of objects which are dynamically extensible.

EP 0 665 493 A2

BACKGROUND OF THE INVENTION**Field of the Invention**

5 The present invention relates to the field of computer systems, and in particular, to programming language compilers, interpreters and related auxiliary systems. More specifically, the present invention relates to extending the functionality of data structures, where the data structures are made up of connected collections of objects, and to mechanisms for projecting such data structures of objects from one type space to another.

10

Background

15 The implementation of modern programming languages, including object oriented programming languages, is generally performed by the use of programs called *Compilers*. A Compiler is a program that reads a program written in one language - the *source* language - and translates it into an equivalent program in another language - the *target* language. A particular example of such Compilers which are used with object oriented programs is the C++ compiler.

20 Shown in **Figure 1** is one embodiment of a compiler 42, comprising a lexical analyzer and parser 44, an intermediate representation builder 46, a semantic analyzer 48, and a code generator 50. These elements are sequentially coupled to each other. Together, they transform program source code 52 into tokenized statements 54, intermediate representations 56, annotated intermediate representations 58, and ultimately intermediate form code 60 with data references made on a symbolic basis. The annotated intermediate representation 58 is preferably referred to as an *Annotated Semantic Graph* (ASG). The lexical analyzer and parser 44, the intermediate representation builder 46, and the semantic analyzer 48 are intended to represent a broad category of these elements found in most compilers. The constitutions and basic functions of these elements are well known and will not be otherwise described further here. Similarly, a variety of well known tokens, intermediate representations, annotations, and intermediate forms may also be used to practice the present invention.

25 During the analysis phase of the compilation, the operations implied by the source program are determined and recorded in a hierarchical structure called a *Syntax Tree* or more generally, an *Abstract Syntax Tree* (AST). Referring now to **Figure 2** an AST for the expression "position := A + B * 60" is illustrated. In this type of tree, each *node* 12, 14, 16 represents an operation, and the children of a node 20, 22, 18 represent the arguments of the operation. (This AST, when modified to describe the semantic relationships of the elements of the tree, is also preferable referred to as an *Annotated Semantic Graph* - (ASG) described above.) Referring now to **Figure 3** the expression "position := A + B * 60" is illustrated as the more general expression "id1 := id2 + id3 * 60 (a number for example)" 30, in **Figure 3(a)**, which can be represented as a data structure 31, for example in an intermediate representation form called a "*three-address code*" form as shown in **Figure 3(b)**. This type of three-address code data structure 31 is like the assembly language for a machine in which every memory location can act like a register. Each three-address code instruction 38 has at most one operator 32 and a left hand operand 34 and a right hand operand 36.

30 For further descriptions of various parsers, intermediate representation builders, semantic analyzers, three-address code representations, and code generators, see A.V.Aho, R.Sethi, and J.D. Ullman, *Compilers; Principles, Techniques, and Tools*, Addison- Wesley, 1986, pp. 25-388, and 463-512, ISBN 0-201-10088-6.

35 In an object oriented system, an object is a component comprising data and operations which can be invoked to manipulate the data. The operations are invoked on the object by sending calls to the object. Each object has an object type. The object type defines the operations that can be performed on objects of that type. The object operations are implemented independent of the objects themselves. Additionally, one 40 object type may inherit the object operations defined and implemented for other object types. For further description of object oriented design and programming techniques see "Object-oriented Software Construction" by Bertrand Meyer, Prentice-Hall 1988 ISBN 0-13-629049-3.

45 In modern object oriented language compilers such as C++, the Annotated Semantic Graphs ("ASGs") and their corresponding data structures are represented as connected collections of objects, 50 wherein each node and leaf of a data structure is an object which has its own internal data structure and some methods. As a result, the source code of a large object oriented program will generate a very large ASG and related data structure (which is itself a very large object-oriented program) as an intermediate representation in the compilation process. Many of such compilers do not expose this ASG to users but

merely use it as the input to the code generator of the compiler for production of the object code of the program being compiled.

The success of any programming language depends to a large extent on the set of tools (in addition to the basic compiler and linker) which enable the construction and modification of application programs, and 5 on the ease and utility with which the tools can be used. Tools such as debuggers, code browsers, code analyzers, style analyzers, and other ad hoc programs can enhance the utility of application programs if such tools are available and easy to use. Additionally, if the construction of new tools is made easy enough, new and powerful tools will appear. In order to enhance the development of new tools and to facilitate the construction of such tools, recent developments in the field have encouraged the exposure of the ASG by 10 modifications to compilers to output the ASG data structure, to store this data structure in an object data base so that it is useable by and available to other programs. See, for example, the paper titled "An Extensible Programming Environment for Modula-3" by Mick Jordan, ACM Proceedings SIGSOFT'90 (Irvine, CA), pages 66-76.

15 The Problem

Even though this ASG in the form of a collection of interconnected objects, of a variety of object types, is made available to tool developers, there are many problems associated with using it.

In an object oriented program it is common to have a collection of interconnected objects, of a variety 20 of types. To manipulate such a collection it is often desirable to be able to extend the functionality of the individual objects, in different ways for different and independent clients (such as tool developers in this case), and possibly for multiple clients at a time. The complete set of potential clients may not be known when the code for the collection is compiled, or when the collection is actually built. Furthermore, it is 25 desirable to be able to extend the functionality of the various objects in a manner which preserves the type compatibility rules. That is, it is desirable to allow an entity to refer to a sub-class but not allow a sub-class to refer to a class. This is referred to below as extending the functionality of the objects in a type-safe manner.

There is another aspect to the problem. Even if it were possible to modify the objects to support all the 30 desired extra functionality, it might not be desirable to do so. To add lots of methods (and the instance data required by their implementation) can lead to an effect called *interface bloat*, in which objects pay the cost of all that functionality whether they use it or not. That is, as users begin to use the ASG objects, they 35 would add methods, which requires a recompilation of the code. As more methods are added, the objects are necessarily made more complex to function, to use, to explain, etc., and require more internal memory space for storage. And as these objects grow bigger through the addition of additional methods, the data base of these objects must be made bigger. Therefore what is needed is a way to use the ASG objects without the necessity for recompilation and without the necessity of using all of the pre-existing methods of the objects if they are not required for the new application.

The problem is not specific to any one programming language, but is exacerbated by the presence of 40 inheritance. This is because inheritance introduces the difference between the static and the dynamic type of an object. In this specification, C++ is used for the examples and for framing a solution to these problems. See for example, "The Annotated C++ Reference Manual" Ellis and Stroustrup, Addison-Wesley 1990, ISBN 0-201-51459-1.

As more fully described below, the present invention is a type-safe framework for dynamically 45 extensible objects which is applicable to the problem of making a large data structure of objects easily accessible to tool developers, but in addition is applicable to any environment for modifying a collection of connected objects of a variety of types.

SUMMARY OF THE INVENTION

50 The present invention provides a system and process for making use of pre-existing data-structures which represent a computer program, in a way which has the advantages of shortening the time and cost required to create a new version of the computer program. The pre-existing data-structure is modified to produce a shadow data-structure which contains shadows of only those elements or nodes of the pre-existing data-structure required to perform the tasks of the new version of the computer program. The 55 present invention includes processes to make the data-structure of the original program shadowable; processes to use data from the original program compilation process in compiling the new version of the program, including processes to create a shadow data-structure; and processes to use the new version of the computer program along with the shadow data-structure to create the desired execution. This new

version of the computer program is typically a tool for checking or observing the original program's execution in some manner. Moreover, the system and processes disclosed provide mechanisms for a software manufacturer to create type-safe versions of a connected collection of objects which are dynamically extensible.

5 A computer implemented method is disclosed in which a first data-structure is provided, and wherein a second data-structure is constructed by using a shadow-map which is operative to create and store a pair of pointers, this second data-structure being related to the first data-structure in some manner. Each element or node in the second data-structure is related to the first data-structure by using the pair of pointers in the shadow-map, wherein for each node in the second data-structure a first pointer of a pointer pair points to
10 the node in the second data-structure and a second pointer of the pointer pair points to the corresponding node in the first data-structure. The method further includes using the second data-structure as an executable variation of the first data-structure. These data structures are collections of connected objects.

15 Also disclosed is a computer implemented method for creating a shadow-data-structure from an existing data-structure by converting the original program source code into code which will produce a shadowable intermediate data structure; creating a shadow node for each node of the shadowable data-structure; connecting the shadow nodes into a shadow data-structure and relating the nodes of the shadow data-structure to the original data structure by means of a shadow-map.

20 Also disclosed and claimed is a system using a computer for dynamically extending the use of a data-structure which consists of a first data-structure, a shadow-map which contains a pair of pointers, a second data-structure which is related to the first data-structure by means of the pointer pairs in the shadow-map, wherein a computer code device can use the second data-structure as an executable variation of the first data-structure.

25 Also disclosed and claimed is a computer mechanism for projecting a first data-structure of objects from one type space into another type space in a type-safe manner, which includes a first program code mechanism operative to convert an original source code into one which produces shadowable data-structures of objects; a shadow-map device to create a shadow data-structure of objects and to relate the shadowable data-structure of objects to a shadow data-structure of objects.

DESCRIPTION OF THE DRAWINGS

30 The objects, features and advantages of the system of the present invention will be apparent from the following description in which:
Figure 1 illustrates an exemplary compiler.
Figure 2 illustrates an exemplary syntax tree.
35 **Figure 3** illustrates a syntax tree (3a) and its corresponding data structure (3b).
Figure 4 illustrates an exemplary computer workstation.
Figure 5 illustrates a simple data structure and its shadow.
Figure 6 illustrates a simple data structure and its shadow, showing the shadow map.
Figure 7 illustrates using back pointers to reduce the size of the shadow data structure.
40 **Figure 8** illustrates shadowing cycles.
Figure 9 illustrates shadowing in the face of inheritance.
Figure 10 illustrates primary classes and methods used to implement shadows.
Figure 11 illustrates a task flow diagram for a first programmer.
Figure 12 illustrates a class structure for an example data structure.
45 **Figure 13** illustrates a task flow diagram for a second programmer.
Figure 14 illustrates a task flow diagram for a program written by the combined efforts of the first and second programmers.
Figure 15 illustrates a flow chart for a subroutine to shadow any node of a shadowable data structure.
Figure 16 illustrates a flow chart for a create_shadow method for a node.
50 **Figure 17** illustrates an exemplary method for a shadow factory to create a shadow node.
Figure 18 illustrates a flow chart for the autodefine program.
Figure 19 illustrates a chart depicting exemplary implementation responsibilities for producing shadow data structures.
Figure 20 illustrates an exemplary inheritance data structure.

NOTATIONS AND NOMENCLATURE

The detailed descriptions which follow may be presented in terms of program procedures executed on a computer or network of computers. These procedural descriptions and representations are the means used by those skilled in the art to most effectively convey the substance of their work to others skilled in the art.

A procedure is here, and generally, conceived to be a self-consistent sequence of steps leading to a desired result. These steps are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It proves convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like. It should be noted, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

Further, the manipulations performed are often referred to in terms, such as adding or comparing, which are commonly associated with mental operations performed by a human operator. No such capability of a human operator is necessary, or desirable in most cases, in any of the operations described herein which form part of the present invention; the operations are machine operations. Useful machines for performing the operations of the present invention include general purpose digital computers or similar devices.

The present invention also relates to apparatus for performing these operations. This apparatus may be specially constructed for the required purposes or it may comprise a general purpose computer as selectively activated or reconfigured by a computer program stored in the computer. The procedures presented herein are not inherently related to a particular computer or other apparatus. Various general purpose machines may be used with programs written in accordance with the teachings herein, or it may prove more convenient to construct more specialized apparatus to perform the required method steps. The required structure for a variety of these machines will appear from the description given.

DESCRIPTION OF THE PREFERRED EMBODIMENT

In an object-oriented program it is common to have a collection of interconnected objects, of a variety of types. To manipulate such a collection it is often desirable to be able to extend the functionality of the individual objects, in different ways for different and independent clients, and possibly for more than one client at a time. The complete set of potential clients may not be known when the code for the collection is compiled, or when the collection is actually built. Furthermore, it is desirable to be able to extend the functionality of the various objects in a type safe manner, that is, allowing an entity to refer to a sub-class but not allowing a sub-class to refer to a class.

'Shadowing' is a flexible way to solve this problem that permits a collection of objects to be projected from one type-space to another. Internally, a simple form of run-time typing is used to provide type-safety. In the preferred embodiment, both the shadow technology and the run-time typing technology use a specialized *ShadowMap* object to create shadow nodes of existing data structures and to maintain a mapping of the relationship of each shadow node to its corresponding node in the original data structure.

A process is disclosed which is a type-safe framework for dynamically extensible objects which is applicable to any environment for modifying a collection of connected objects of a variety of types, without having to inherit all of the methods of the original objects. The disclosed apparatus and process permits a user (such as a tool developer, for example) to create a shadow data-structure of an existing data-structure for subsequent execution, where both data-structures are connected collections of object-oriented programming objects. The implementation of the invention, while it may be used in any relevant context with any computer program product, is described in the context of a particular type of object-oriented compiler (that is, C++) and exemplary object-oriented programming systems for illustrative purposes. However, no specific knowledge of the illustrated system is required by those skilled in these arts to understand and implement the process and system described in this disclosure, using various other compilers and related tools.

Operating Environment

The environment in which the present invention is used encompasses the general distributed computing system, wherein general purpose computers, workstations, or personal computers are connected via communication links of various types, in a client-server arrangement, wherein programs and data, many in

the form of objects, are made available and shared by various members of the system for execution and access by other members of the system. Some of the elements of a general purpose workstation computer 20 are shown in **Figure 4**, wherein a processor 1 is shown, having an Input/output ("I/O") section 2, a central processing unit ("CPU") 3 and a memory section 4. The I/O section 2 is connected to a keyboard 5, a display unit 6, a disk storage unit 9 and a CD-ROM drive unit 7. The CD-ROM unit 7 can read a CD-ROM medium 8 which typically contains programs 10 and data. A computer display icon 11 is shown on the display unit 6. Similar workstations may be connected by a communications path to form a distributed computer system.

The present invention will be discussed below in terms of basic background and underlying technology 10 to provide a summary of how to build the invention and how it functions, followed by a detailed description of how to use the invention.

Underlying Technology

15 Shadows

Shadows are based on a mechanism which make it possible to project a data structure from one type space into another. The primary reason to do this is to add extra functionality to the nodes of the data structure, such as extra methods. More sophisticated use of the mechanisms may change the actual shape 20 of the shadow type structure itself.

One particular feature of shadows is that they are designed to work well even when inheritance is used for the nodes of the data structures. Support is also provided for lazily evaluating the projection of the data structure.

Referring now to **Figure 5**, a simple data structure and its shadow are depicted 90. On the left is a 25 simple data structure 92, involving nodes of type A 96, B 97, and C 99. On the right is the shadow data structure 94, in which the nodes are of type A' 106, B' 107, and C' 109. The shadow may perhaps have been created in order to extend the types of the nodes in the data structure, and because it was not possible to change the data structure itself. In **Figure 5**, node A 96, is connected to node B 97, by the pointer 98, and node C 99 is connected to nodes A 96, and B 97, by pointers 100 and 102 respectively. 30 The corresponding nodes in the shadow data structure 94 are similarly connected by pointers 108, 110 and 112.

In **Figure 5**, there is a 1-1 correspondence between the instances of the nodes in the data structure and its shadow, and there is a 1-1 correspondence between the types of the nodes in the data structure and the types of the nodes in its shadow. As will be indicated, neither of these conditions are required.

35 Note also the convention that is used throughout this disclosure: that T' stands for the type of the shadow of a node of type T.

The Shadow Map

40 In the preferred embodiment, the shadow data-structure illustrated in **Figure 5** is created by an object, which again in the preferred embodiment is called a *ShadowMap* object. Those skilled in the art will recognize that there are numerous ways to perform these same create functions.

The *ShadowMap* object provides a lazily evaluated mapping from the nodes in the original data structure to the corresponding nodes in the shadow data structure. Lazy evaluation means that if no one 45 ever asks for a particular node to be shadowed the shadow node will never be created. Entries in the mapping are generated on demand by calling upon methods to fabricate nodes in the shadow data structure. These methods are called the *factory methods* of the map. Factory methods are supplied by a client that wishes to create a shadow structure.

Referring now to **Figure 6**, a data structure 92 and its shadow data structure 94, as illustrated in **Figure 5** are depicted again, but this time a shadow map 122, which is used in the creation of the shadow data structure 94, is also illustrated. The shadow map 122 is created by the *ShadowMap* object and includes the pointer pairs 124, 130; 126, 132 and 128, 134 which are retained in the shadow map 122, and which relate each node in the original data structure 92 to its corresponding shadow node in the shadow data structure 94.

55 Thus, to get the shadow of a node, it is sufficient to ask the *ShadowMap* object for the shadow, passing it the node in question. Internally, the *ShadowMap* object sees if it already contains the shadow node for the requested node, by checking the shadow map 122. If it does, that value is immediately returned; if not, it calls a factory method to create the shadow node. The result that is returned from the factory method is

both remembered by the `ShadowMap` object for future reference (by storing a pointer in the shadow map **122**) and also returned to the caller.

Once a shadow data structure has been created, the shadow map **122** often plays no further role and might even be deleted. Alternatively, it does contain pointers to all the nodes in the shadow data structure, 5 and so can prove helpful in performing operations involving the complete set of shadow nodes, such as deleting them when they are no longer needed. Those skilled in the art will realize that the shadow data structure and shadow map may be used for any number of alternative applications.

Manipulating the data structure

10 In the examples above, an unwritten implication of the figures is that the nodes of the shadow data structure are very similar to the nodes in the original, merely having the types projected into the shadow type space. However, there is no reason for this to be so; the methods on the `ShadowMap` object that generate shadow nodes can generate whatever form of shadow nodes they choose. The only restriction placed on these methods is that for each node in the original data structure, there should be a single 15 corresponding node in the shadow data structure. However, a node in the original data structure can have no corresponding shadow node, in which case the shadow map will contain a null pointer in the appropriate entry.

20 For example, suppose that we are merely creating the shadow in order to add a method to each of the nodes, such as a "print" method. In this case, the shadow data structure might choose merely to store back pointers to the nodes in the original data structure, without shadowing the relationship between the nodes in the shadow space at all. The print method can get at the instance data to be printed by following the back pointer to the nodes being shadowed; if needed, each shadow node can get at its shadow children by re-evaluating the shadow of the children in the original data structure on demand. This type of shadow data 25 structure is illustrated in **Figure 7**, wherein the original data structure **92** is shown, and the shadow map **122**, but a new shadow data structure **142** is shown containing only nodes **144**, **146** and **148** which are back pointers to the corresponding nodes of the original data structure **92**.

25 As well as specifying the instance data for the shadow nodes, the factory methods may also change the type of the nodes on a per-instance basis. Thus two nodes in the original data structure may be 30 represented by the same type, albeit with different instance data. In shadowing these two nodes, a factory method may examine the instance data and choose between two different representations for the shadowed node.

35 For example, in the case where the data structure being shadowed is a parse tree generated by a compiler front end the following could obtain. A binary expression node in the parse tree might be represented by a single node type, `PT_BinaryExpr`, with an enumerated value indicating the operation of the expression: add, subtract, multiply or divide. However, a compiler back end may choose to shadow this one type of node by generating different types of shadow nodes depending on the operation involved, such as `CG_AddExpr`, `CG_SubExpr`, `CG_MultExpr`, or `CGDivExpr`.

40 It may also be the case that not all of a data structure needs to be shadowed. Consider again the case of the parse tree generated by a compiler front end. An incremental code-generator may only need to generate code for (and hence shadow) the parse tree for one or two methods and not the whole tree. Those skilled in these arts will recognize that the present invention may be used to generate any number of kinds of shadow data structures for various situations.

45 Cyclic and acyclic data structures

In order to shadow a node, the preferred embodiment described so far says that a client should simply ask the `ShadowMap` object for the shadow, and that the `ShadowMap` object will compute it if necessary by calling the appropriate factory method. However, if invoked, that method will often call upon the `ShadowMap` 50 object for the shadows of child nodes of the original, and this may in turn recursively invoke the factory method. All will be well, provided the data structure being shadowed is acyclic.

55 In the preferred embodiment, if the data structure contains cycles, slightly more care must be taken. Specifically, the (possibility of a) cycle must be known and the cycle explicitly broken. Referring now to **Figure 8**, a data structure **152** containing a cycle structure is shown. That is, node A **162** is connected to node B **164** which is connected to node C **166**, which is connected to node A **162**. In creating the shadow data structure **154**, in the course of shadowing A **162**, B **164** needs to be shadowed, which needs C **166** to be shadowed, which potentially needs A **162** to be shadowed—and which may not have been completed yet, and so will not be available in the shadow map. Instead, the shadow node C' **172**, created by the

ShadowMap object, stores a lazy pointer 178 to shadow node A' 168 which is only dereferenced to a real pointer when absolutely necessary. In particular, it can only be dereferenced after the shadowing of A 162 has been completed, in which case the ShadowMap object can provide the information that the result of shadowing A 162 is A' 168. Once it has been converted to a real pointer, that value is remembered so that 5 future accesses will be almost as fast as using the pointer itself.

In the C ++ implementation of the preferred embodiment, it serves to use a special type of pointer some where within the cycle, which is lazily evaluated. This is described in detail below in the section describing, "Lazy shadows for cyclic data."

10 **How to Make and Use the Invention.**

C++ Implementation in the Preferred Embodiment

15 In the preferred embodiment of the present invention, implementing a shadow map by calling upon a factory object and caching the results in a hash table becomes complicated when inheritance is taken into account, and when constrained by the rules and idiosyncrasies of the C ++ type system.

Inheritance

20 Referring now to **Figure 9**, a data structure 181 and its shadow data structure 183 illustrate shadowing in the face of inheritance. In the data structure 181, consider two classes, B and D, such that B is a base class and D is derived from B. Now consider another class, X, which contains a pointer of declared type B* 182. When that pointer is initialized, it could be initialized with a pointer to an object of class B 188, or it could be initialized with a pointer to an object of class D 184, with the pointer being widened from type D to 25 type B*.

Now consider the shadow type space 183. Suppose B' 196 is the shadow type for B 188, D' 192 is the shadow node for D 184 and node X' 190, 194 for X 182, 186. Then, in a simple shadow mapping, node X' 190 will contain a pointer of type B'*191, and that pointer may point to objects of type B' 196, or D' 192 if the pointer has been widened.

30 The problem, then, is to correctly shadow not only the dynamic (or 'true') type of the objects, but also the static (or 'perceived') type.

Shadowable, ShadowMap and T'::shadow.

35 In the preferred embodiment, a number of classes and methods collaborate to provide the shadowing mechanism. **Figure 10** outlines the classes and primary methods involved.

Shadowable

40 Nodes to be shadowed must inherit from Shadowable. This provides an important virtual method that is used during the process of instantiating the shadow of a node.

```
class Shadowable
45
{
public:
    virtual Narrowable *create_shadow (ShadowMap *map) = 0;
50
};
```

55 Although all nodes to be shadowed must provide an implementation of create_shadow, this method is never directly called by a client. In the preferred embodiment, the implementation is boiler-plate code that invokes the appropriate factory method on an appropriate subtype of the shadow map argument. This boiler-plate code can be mechanically generated by the program "autodefine". The map argument provides a context for the shadow, permitting different shadows of the same node to coexist.

ShadowMap

In the preferred embodiment, the object `ShadowMap` represents the shadow map. It has one public method, to look up a shadow node in the map, and to create one if not found. However, even this method is not normally called directly by clients. Instead, clients should call the shadow method on the target class instead, which provides more convenient and type-safe access to shadow nodes.

There will normally be two levels of inheritance based on `ShadowMap`. The first extends the `ShadowMap` with a set of pure virtual methods that define the ability to shadow specific types of nodes; the second level provides the implementation of those pure virtual methods.

10

```
class ShadowMap
{
public:
    Narrowable *lookup_or_create_shadow(Shadowable *orig);
    // returns the shadow node for 'orig'.
};
```

20

25

T'::shadow

30 The most important method in the preferred embodiment is `T'::shadow`, since it is the primary way a client creates or accesses the shadow of a node. It is a static method on the desired type, `T'`, of the shadow node. It takes two arguments: the node to be shadowed, of type `T`, and a shadow map.

```
static T' *T'::shadow(T *t, ShadowMap *map);
```

35 It returns the shadow "t" according to "map." The argument passed as "map" should be an instance of a map capable of generating `T'` nodes from `T` nodes. This can be enforced by using an appropriate subtype of `ShadowMap`, as described below.

Making a data structure shadowable

40 The steps that must be taken to make a data structure shadowable in the preferred embodiment are now described. Referring now to **Figure 11**, a task flow diagram 200 for the programmer of the original data structure is illustrated. In **Figure 11** the process begins with the programmer writing header files 204 to describe the objects that go to make up the data structure. These header files are augmented to add "create_shadow" method to each type of shadowable object 206. The objects declared in the header files will inherit from the library class "Shadowable". Since the data structure is composed of objects, and objects have methods, the programmer then writes the code 208 for the methods of the objects in the data structure. In the preferred embodiment, some of the methods can be mechanically generated by running the header files through the program "autodefine" to create definitions and boiler plate functions and objects 210. The output of "autodefine" is C++ code that is combined with the written code from step 208.

45 The programmer must also provide a special "ShadowMap" object for others who may wish to shadow the data structure. The "ShadowMap" class is also generated by "autodefine" in step 210 resulting in some header files that are combined with the header files from step 204 and all of these header files are collected together for later use by others 212. All of the header files and code files are then processed by a compiler 216 to generate object code. The output from the compiler 221 will be used by both the originating

50 programmer and by others who may wish to create shadow data structures therefrom 218.

In the preferred embodiment, some additional comments are of interest. To make a data structure shadowable, all the nodes to be shadowed in the data structure must inherit from a common root class, and this common root class must itself inherit from "Shadowable." The need for a common root class other than

"Shadowable" is a requirement of the program "autodefine" which is used to automatically generate parts of the shadow machinery.

All nodes to be shadowed must implement the abstract "create_shadow" method inherited from "Shadowable."

5 Narrowable *create_shadow (ShadowMap *);

These implementations of "create_shadow" are typically declared as private methods on the nodes to be shadowed, to emphasize that they are not to be called directly by clients. (The method is declared public on "Shadowable.") The implementation is boiler-plate code that invokes the appropriate factory method on an appropriate subtype of the shadow map argument. This boiler-plate code can be mechanically generated by the utility "autodefine".

10 One more thing must be done to make a data structure shadowable: an abstract subtype of "ShadowMap" must be provided that defines a special method for each node type, T, to be shadowed:

 Narrowable *shadowT(T *);

These methods will be referred to in the boiler-plate implementations of "create_shadow."

15

Example

Referring now to **Figure 12**, a class structure for an example data structure **230** is shown. The data structure containing nodes of type **A 238**, **B 234**, **C 240** and **D 236**. Suppose **C 240** inherits from **A 238** and

20 **D 236** inherits from both **A 238**, and **B 234**. Let us suppose that "Roman" **232** is the supertype for these four classes.

Then we would expect to see the following patterns in the code:

25

30

35

40

45

50

55

```
5      class Roman: public virtual Shadowable
6      {
7          public:
8
9              //  any generic methods on all Roman node types can be declared here
10
11         private:
12             Narrowable *create_shadow (ShadowMap *);
13
14     };
15
16     class A: public virtual Roman
17     {
18
19         public:
20
21             //  A methods...
22
23         private:
24             Narrowable *create_shadow (ShadowMap *);
25
26     };
27
28     class B: public virtual Roman
29     {
30
31         public:
32
33             //  B methods...
34
35         private:
36             Narrowable *create_shadow (ShadowMap *);
37
38     };
39
40     class C: public virtual B
41     {
42
43         public:
44
45             //  C methods...
46
47     };
48
49     class D: public virtual C
50     {
51
52         public:
53
54             //  D methods...
55
56     };
57
58     class E: public virtual D
59     {
60
61         public:
62
63             //  E methods...
64
65     };
66
67     class F: public virtual E
68     {
69
70         public:
71
72             //  F methods...
73
74     };
75
76     class G: public virtual F
77     {
78
79         public:
80
81             //  G methods...
82
83     };
84
85     class H: public virtual G
86     {
87
88         public:
89
90             //  H methods...
91
92     };
93
94     class I: public virtual H
95     {
96
97         public:
98
99             //  I methods...
100
101    };
102
103    class J: public virtual I
104    {
105
106        public:
107
108            //  J methods...
109
110    };
111
112    class K: public virtual J
113    {
114
115        public:
116
117            //  K methods...
118
119    };
120
121    class L: public virtual K
122    {
123
124        public:
125
126            //  L methods...
127
128    };
129
130    class M: public virtual L
131    {
132
133        public:
134
135            //  M methods...
136
137    };
138
139    class N: public virtual M
140    {
141
142        public:
143
144            //  N methods...
145
146    };
147
148    class O: public virtual N
149    {
150
151        public:
152
153            //  O methods...
154
155    };
156
157    class P: public virtual O
158    {
159
160        public:
161
162            //  P methods...
163
164    };
165
166    class Q: public virtual P
167    {
168
169        public:
170
171            //  Q methods...
172
173    };
174
175    class R: public virtual Q
176    {
177
178        public:
179
180            //  R methods...
181
182    };
183
184    class S: public virtual R
185    {
186
187        public:
188
189            //  S methods...
190
191    };
192
193    class T: public virtual S
194    {
195
196        public:
197
198            //  T methods...
199
200    };
201
202    class U: public virtual T
203    {
204
205        public:
206
207            //  U methods...
208
209    };
210
211    class V: public virtual U
212    {
213
214        public:
215
216            //  V methods...
217
218    };
219
220    class W: public virtual V
221    {
222
223        public:
224
225            //  W methods...
226
227    };
228
229    class X: public virtual W
230    {
231
232        public:
233
234            //  X methods...
235
236    };
237
238    class Y: public virtual X
239    {
240
241        public:
242
243            //  Y methods...
244
245    };
246
247    class Z: public virtual Y
248    {
249
250        public:
251
252            //  Z methods...
253
254    };
255
256    class AA: public virtual Z
257    {
258
259        public:
260
261            //  AA methods...
262
263    };
264
265    class BB: public virtual AA
266    {
267
268        public:
269
270            //  BB methods...
271
272    };
273
274    class CC: public virtual BB
275    {
276
277        public:
278
279            //  CC methods...
280
281    };
282
283    class DD: public virtual CC
284    {
285
286        public:
287
288            //  DD methods...
289
290    };
291
292    class EE: public virtual DD
293    {
294
295        public:
296
297            //  EE methods...
298
299    };
299
300    class FF: public virtual EE
301    {
302
303        public:
304
305            //  FF methods...
306
307    };
308
309    class GG: public virtual FF
309    {
310
311        public:
312
313            //  GG methods...
314
315    };
316
317    class HH: public virtual GG
318    {
319
320        public:
321
322            //  HH methods...
323
324    };
325
326    class II: public virtual HH
327    {
328
329        public:
330
331            //  II methods...
332
333    };
334
335    class JJ: public virtual II
336    {
337
338        public:
339
340            //  JJ methods...
341
342    };
343
344    class KK: public virtual JJ
345    {
346
347        public:
348
349            //  KK methods...
350
351    };
352
353    class LL: public virtual KK
354    {
355
356        public:
357
358            //  LL methods...
359
360    };
361
362    class MM: public virtual LL
363    {
364
365        public:
366
367            //  MM methods...
368
369    };
370
371    class NN: public virtual MM
372    {
373
374        public:
375
376            //  NN methods...
377
378    };
379
380    class OO: public virtual NN
381    {
382
383        public:
384
385            //  OO methods...
386
387    };
388
389    class PP: public virtual OO
390    {
391
392        public:
393
394            //  PP methods...
395
396    };
397
398    class QQ: public virtual PP
399    {
400
401        public:
402
403            //  QQ methods...
404
405    };
406
407    class RR: public virtual QQ
408    {
409
410        public:
411
412            //  RR methods...
413
414    };
415
416    class SS: public virtual RR
417    {
418
419        public:
420
421            //  SS methods...
422
423    };
424
425    class TT: public virtual SS
426    {
427
428        public:
429
430            //  TT methods...
431
432    };
433
434    class UU: public virtual TT
435    {
436
437        public:
438
439            //  UU methods...
440
441    };
442
443    class VV: public virtual UU
444    {
445
446        public:
447
448            //  VV methods...
449
450    };
451
452    class WW: public virtual VV
453    {
454
455        public:
456
457            //  WW methods...
458
459    };
460
461    class XX: public virtual WW
462    {
463
464        public:
465
466            //  XX methods...
467
468    };
469
470    class YY: public virtual XX
471    {
472
473        public:
474
475            //  YY methods...
476
477    };
478
479    class ZZ: public virtual YY
480    {
481
482        public:
483
484            //  ZZ methods...
485
486    };
487
488    class AAA: public virtual ZZ
489    {
490
491        public:
492
493            //  AAA methods...
494
495    };
496
497    class BBB: public virtual AAA
498    {
499
500        public:
501
502            //  BBB methods...
503
504    };
505
506    class CCC: public virtual BBB
507    {
508
509        public:
510
511            //  CCC methods...
512
513    };
514
515    class DDD: public virtual CCC
516    {
517
518        public:
519
520            //  DDD methods...
521
522    };
523
524    class EEE: public virtual DDD
525    {
526
527        public:
528
529            //  EEE methods...
530
531    };
532
533    class FFF: public virtual EEE
534    {
535
536        public:
537
538            //  FFF methods...
539
540    };
541
542    class GGG: public virtual FFF
543    {
544
545        public:
546
547            //  GGG methods...
548
549    };
550
551    class HHH: public virtual GGG
552    {
553
554        public:
555
556            //  HHH methods...
557
558    };
559
560    class III: public virtual HHH
561    {
562
563        public:
564
565            //  III methods...
566
567    };
568
569    class JJJ: public virtual III
570    {
571
572        public:
573
574            //  JJJ methods...
575
576    };
577
578    class KKK: public virtual JJJ
579    {
580
581        public:
582
583            //  KKK methods...
584
585    };
586
587    class LLL: public virtual KKK
588    {
589
590        public:
591
592            //  LLL methods...
593
594    };
595
596    class MMM: public virtual LLL
597    {
598
599        public:
600
601            //  MMM methods...
602
603    };
604
605    class NNN: public virtual MMM
606    {
607
608        public:
609
610            //  NNN methods...
611
612    };
613
614    class OOO: public virtual NNN
615    {
616
617        public:
618
619            //  OOO methods...
620
621    };
622
623    class PPP: public virtual OOO
624    {
625
626        public:
627
628            //  PPP methods...
629
630    };
631
632    class QQQ: public virtual PPP
633    {
634
635        public:
636
637            //  QQQ methods...
638
639    };
640
641    class RRR: public virtual QQQ
642    {
643
644        public:
645
646            //  RRR methods...
647
648    };
649
650    class SSS: public virtual RRR
651    {
652
653        public:
654
655            //  SSS methods...
656
657    };
658
659    class TTT: public virtual SSS
660    {
661
662        public:
663
664            //  TTT methods...
665
666    };
667
668    class UUU: public virtual TTT
669    {
670
671        public:
672
673            //  UUU methods...
674
675    };
676
677    class VVV: public virtual UUU
678    {
679
680        public:
681
682            //  VVV methods...
683
684    };
685
686    class WWW: public virtual VVV
687    {
688
689        public:
690
691            //  WWW methods...
692
693    };
694
695    class XXX: public virtual WWW
696    {
697
698        public:
699
700            //  XXX methods...
701
702    };
703
704    class YY: public virtual XXX
705    {
706
707        public:
708
709            //  YY methods...
710
711    };
712
713    class ZZ: public virtual YY
714    {
715
716        public:
717
718            //  ZZ methods...
719
720    };
721
722    class AAAA: public virtual ZZ
723    {
724
725        public:
726
727            //  AAAA methods...
728
729    };
730
731    class BBBB: public virtual AAAA
732    {
733
734        public:
735
736            //  BBBB methods...
737
738    };
739
740    class CCCC: public virtual BBBB
741    {
742
743        public:
744
745            //  CCCC methods...
746
747    };
748
749    class DDDD: public virtual CCCC
750    {
751
752        public:
753
754            //  DDDD methods...
755
756    };
757
758    class EEEE: public virtual DDDD
759    {
760
761        public:
762
763            //  EEEE methods...
764
765    };
766
767    class FFFF: public virtual EEEE
768    {
769
770        public:
771
772            //  FFFF methods...
773
774    };
775
776    class GGGG: public virtual FFFF
777    {
778
779        public:
780
781            //  GGGG methods...
782
783    };
784
785    class HHHH: public virtual GGGG
786    {
787
788        public:
789
790            //  HHHH methods...
791
792    };
793
794    class IIII: public virtual HHHH
795    {
796
797        public:
798
799            //  IIII methods...
800
801    };
802
803    class JJJJ: public virtual IIII
804    {
805
806        public:
807
808            //  JJJJ methods...
809
810    };
811
812    class KKKK: public virtual JJJJ
813    {
814
815        public:
816
817            //  KKKK methods...
818
819    };
820
821    class LLLL: public virtual KKKK
822    {
823
824        public:
825
826            //  LLLL methods...
827
828    };
829
830    class MLLL: public virtual LLLL
831    {
832
833        public:
834
835            //  MLLL methods...
836
837    };
838
839    class NLLL: public virtual MLLL
840    {
841
842        public:
843
844            //  NLLL methods...
845
846    };
847
848    class OLLL: public virtual NLLL
849    {
850
851        public:
852
853            //  OLLL methods...
854
855    };
856
857    class PPPP: public virtual OLLL
858    {
859
860        public:
861
862            //  PPPP methods...
863
864    };
865
866    class QQQQ: public virtual PPPP
867    {
868
869        public:
870
871            //  QQQQ methods...
872
873    };
874
875    class RRRR: public virtual QQQQ
876    {
877
878        public:
879
880            //  RRRR methods...
881
882    };
883
884    class SSSS: public virtual RRRR
885    {
886
887        public:
888
889            //  SSSS methods...
890
891    };
892
893    class TTTT: public virtual SSSS
894    {
895
896        public:
897
898            //  TTTT methods...
899
900    };
901
902    class UUUU: public virtual TTTT
903    {
904
905        public:
906
907            //  UUUU methods...
908
909    };
910
911    class VVVV: public virtual UUUU
912    {
913
914        public:
915
916            //  VVVV methods...
917
918    };
919
920    class WWWW: public virtual VVVV
921    {
922
923        public:
924
925            //  WWWW methods...
926
927    };
928
929    class XXXX: public virtual WWWW
930    {
931
932        public:
933
934            //  XXXX methods...
935
936    };
937
938    class YY: public virtual XXXX
939    {
940
941        public:
942
943            //  YY methods...
944
945    };
946
947    class ZZ: public virtual YY
948    {
949
950        public:
951
952            //  ZZ methods...
953
954    };
955
956    class AAAAA: public virtual ZZ
957    {
958
959        public:
960
961            //  AAAAA methods...
962
963    };
964
965    class BBBBB: public virtual AAAAA
966    {
967
968        public:
969
970            //  BBBBB methods...
971
972    };
973
974    class CCCCC: public virtual BBBBB
975    {
976
977        public:
978
979            //  CCCCC methods...
980
981    };
982
983    class DDDDD: public virtual CCCCC
984    {
985
986        public:
987
988            //  DDDDD methods...
989
990    };
991
992    class EEEEE: public virtual DDDDD
993    {
994
995        public:
996
997            //  EEEEE methods...
998
999    };
999
1000   class FFFFF: public virtual EEEEE
1001  {
1002
1003     public:
1004
1005        //  FFFFF methods...
1006
1007  };
1008
1009  class GGGGG: public virtual FFFFF
1010 {
1011
1012     public:
1013
1014        //  GGGGG methods...
1015
1016  };
1017
1018  class HHHHH: public virtual GGGGG
1019 {
1020
1021     public:
1022
1023        //  HHHHH methods...
1024
1025  };
1026
1027  class IIIII: public virtual HHHHH
1028 {
1029
1030     public:
1031
1032        //  IIIII methods...
1033
1034  };
1035
1036  class JJJJJ: public virtual IIIII
1037 {
1038
1039     public:
1040
1041        //  JJJJJ methods...
1042
1043  };
1044
1045  class KKKKK: public virtual JJJJJ
1046 {
1047
1048     public:
1049
1050        //  KKKKK methods...
1051
1052  };
1053
1054  class LLLL: public virtual KKKKK
1055 {
1056
1057     public:
1058
1059        //  LLLL methods...
1060
1061  };
1062
1063  class MLLL: public virtual LLLL
1064 {
1065
1066     public:
1067
1068        //  MLLL methods...
1069
1070  };
1071
1072  class NLLL: public virtual MLLL
1073 {
1074
1075     public:
1076
1077        //  NLLL methods...
1078
1079  };
1080
1081  class OLLL: public virtual NLLL
1082 {
1083
1084     public:
1085
1086        //  OLLL methods...
1087
1088  };
1089
1090  class PPPP: public virtual OLLL
1091 {
1092
1093     public:
1094
1095        //  PPPP methods...
1096
1097  };
1098
1099  class QQQQ: public virtual PPPP
1100 {
1101
1102     public:
1103
1104        //  QQQQ methods...
1105
1106  };
1107
1108  class RRRR: public virtual QQQQ
1109 {
1110
1111     public:
1112
1113        //  RRRR methods...
1114
1115  };
1116
1117  class SSSS: public virtual RRRR
1118 {
1119
1120     public:
1121
1122        //  SSSS methods...
1123
1124  };
1125
1126  class TTTT: public virtual SSSS
1127 {
1128
1129     public:
1130
1131        //  TTTT methods...
1132
1133  };
1134
1135  class UUUU: public virtual TTTT
1136 {
1137
1138     public:
1139
1140        //  UUUU methods...
1141
1142  };
1143
1144  class VVVV: public virtual UUUU
1145 {
1146
1147     public:
1148
1149        //  VVVV methods...
1150
1151  };
1152
1153  class WWWW: public virtual VVVV
1154 {
1155
1156     public:
1157
1158        //  WWWW methods...
1159
1160  };
1161
1162  class XXXX: public virtual WWWW
1163 {
1164
1165     public:
1166
1167        //  XXXX methods...
1168
1169  };
1170
1171  class YY: public virtual XXXX
1172 {
1173
1174     public:
1175
1176        //  YY methods...
1177
1178  };
1179
1180  class ZZ: public virtual YY
1181 {
1182
1183     public:
1184
1185        //  ZZ methods...
1186
1187  };
1188
1189  class AAAAA: public virtual ZZ
1190 {
1191
1192     public:
1193
1194        //  AAAAA methods...
1195
1196  };
1197
1198  class BBBBB: public virtual AAAAA
1199 {
1200
1201     public:
1202
1203        //  BBBBB methods...
1204
1205  };
1206
1207  class CCCCC: public virtual BBBBB
1208 {
1209
1210     public:
1211
1212        //  CCCCC methods...
1213
1214  };
1215
1216  class DDDDD: public virtual CCCCC
1217 {
1218
1219     public:
1220
1221        //  DDDDD methods...
1222
1223  };
1224
1225  class EEEEE: public virtual DDDDD
1226 {
1227
1228     public:
1229
1230        //  EEEEE methods...
1231
1232  };
1233
1234  class FFFFF: public virtual EEEEE
1235 {
1236
1237     public:
1238
1239        //  FFFFF methods...
1240
1241  };
1242
1243  class GGGGG: public virtual FFFFF
1244 {
1245
1246     public:
1247
1248        //  GGGGG methods...
1249
1250  };
1251
1252  class HHHHH: public virtual GGGGG
1253 {
1254
1255     public:
1256
1257        //  HHHHH methods...
1258
1259  };
1260
1261  class IIIII: public virtual HHHHH
1262 {
1263
1264     public:
1265
1266        //  IIIII methods...
1267
1268  };
1269
1270  class JJJJJ: public virtual IIIII
1271 {
1272
1273     public:
1274
1275        //  JJJJJ methods...
1276
1277  };
1278
1279  class KKKKK: public virtual JJJJJ
1280 {
1281
1282     public:
1283
1284        //  KKKKK methods...
1285
1286  };
1287
1288  class LLLL: public virtual KKKKK
1289 {
1290
1291     public:
1292
1293        //  LLLL methods...
1294
1295  };
1296
1297  class MLLL: public virtual LLLL
1298 {
1299
1300     public:
1301
1302        //  MLLL methods...
1303
1304  };
1305
1306  class NLLL: public virtual MLLL
1307 {
1308
1309     public:
1310
1311        //  NLLL methods...
1312
1313  };
1314
1315  class OLLL: public virtual NLLL
1316 {
1317
1318     public:
1319
1320        //  OLLL methods...
1321
1322  };
1323
1324  class PPPP: public virtual OLLL
1325 {
1326
1327     public:
1328
1329        //  PPPP methods...
1330
1331  };
1332
1333  class QQQQ: public virtual PPPP
1334 {
1335
1336     public:
1337
1338        //  QQQQ methods...
1339
1340  };
1341
1342  class RRRR: public virtual QQQQ
1343 {
1344
1345     public:
1346
1347        //  RRRR methods...
1348
1349  };
1350
1351  class SSSS: public virtual RRRR
1352 {
1353
1354     public:
1355
1356        //  SSSS methods...
1357
1358  };
1359
1360  class TTTT: public virtual SSSS
1361 {
1362
1363     public:
1364
1365        //  TTTT methods...
1366
1367  };
1368
1369  class UUUU: public virtual TTTT
1370 {
1371
1372     public:
1373
1374        //  UUUU methods...
1375
1376  };
1377
1378  class VVVV: public virtual UUUU
1379 {
1380
1381     public:
1382
1383        //  VVVV methods...
1384
1385  };
1386
1387  class WWWW: public virtual VVVV
1388 {
1389
1390     public:
1391
1392        //  WWWW methods...
1393
1394  };
1395
1396  class XXXX: public virtual WWWW
1397 {
1398
1399     public:
1400
1401        //  XXXX methods...
1402
1403  };
1404
1405  class YY: public virtual XXXX
1406 {
1407
1408     public:
1409
1410        //  YY methods...
1411
1412  };
1413
1414  class ZZ: public virtual YY
1415 {
1416
1417     public:
1418
1419        //  ZZ methods...
1420
1421  };
1422
1423  class AAAAA: public virtual ZZ
1424 {
1425
1426     public:
1427
1428        //  AAAAA methods...
1429
1430  };
1431
1432  class BBBBB: public virtual AAAAA
1433 {
1434
1435     public:
1436
1437        //  BBBBB methods...
1438
1439  };
1440
1441  class CCCCC: public virtual BBBBB
1442 {
1443
1444     public:
1445
1446        //  CCCCC methods...
1447
1448  };
1449
1450  class DDDDD: public virtual CCCCC
1451 {
1452
1453     public:
1454
1455        //  DDDDD methods...
1456
1457  };
1458
1459  class EEEEE: public virtual DDDDD
1460 {
1461
1462     public:
1463
1464        //  EEEEE methods...
1465
1466  };
1467
1468  class FFFFF: public virtual EEEEE
1469 {
1470
1471     public:
1472
1473        //  FFFFF methods...
1474
1475  };
1476
1477  class GGGGG: public virtual FFFFF
1478 {
1479
1480     public:
1481
1482        //  GGGGG methods...
1483
1484  };
1485
1486  class HHHHH: public virtual GGGGG
1487 {
1488
1489     public:
1490
1491        //  HHHHH methods...
1492
1493  };
1494
1495  class IIIII: public virtual HHHHH
1496 {
1497
1498     public:
1499
1500        //  IIIII methods...
1501
1502  };
1503
1504  class JJJJJ: public virtual IIIII
1505 {
1506
1507     public:
1508
1509        //  JJJJJ methods...
1510
1511  };
1512
1513  class KKKKK: public virtual JJJJJ
1514 {
1515
1516     public:
1517
1518        //  KKKKK methods...
1519
1520  };
1521
1522  class LLLL: public virtual KKKKK
1523 {
1524
1525     public:
1526
1527        //  LLLL methods...
1528
1529  };
1530
1531  class MLLL: public virtual LLLL
1532 {
1533
1534     public:
1535
1536        //  MLLL methods...
1537
1538  };
1539
1540  class NLLL: public virtual MLLL
1541 {
1542
1543     public:
1544
1545        //  NLLL methods...
1546
1547  };
1548
1549  class OLLL: public virtual NLLL
1550 {
1551
1552     public:
1553
1554        //  OLLL methods...
1555
1556  };
1557
1558  class PPPP: public virtual OLLL
1559 {
1560
1561     public:
1562
1563        //  PPPP methods...
1564
1565  };
1566
1567  class QQQQ: public virtual PPPP
1568 {
1569
1570     public:
1571
1572        //  QQQQ methods...
1573
1574  };
1575
1576  class RRRR: public virtual QQQQ
1577 {
1578
1579     public:
1580
1581        //  RRRR methods...
1582
1583  };
1584
1585  class SSSS: public virtual RRRR
1586 {
1587
1588     public:
1589
1590        //  SSSS methods...
1591
1592  };
1593
1594  class TTTT: public virtual SSSS
1595 {
1596
1597     public:
1598
1599        //  TTTT methods...
1600
1601  };
1602
1603  class UUUU: public virtual TTTT
1604 {
1605
1606     public:
1607
1608        //  UUUU methods...
1609
1610  };
1611
1612  class VVVV: public virtual UUUU
1613 {
1614
1615     public:
1616
1617        //  VVVV methods...
1618
1619  };
1620
1621  class WWWW: public virtual VVVV
1622 {
1623
1624     public:
1625
1626        //  WWWW methods...
1627
1628  };
1629
1630  class XXXX: public virtual WWWW
1631 {
1632
1633     public:
1634
1635        //  XXXX methods...
1636
1637  };
1638
1639  class YY: public virtual XXXX
1640 {
1641
1642     public:
1643
1644        //  YY methods...
1645
1646  };
1647
1648  class ZZ: public virtual YY
1649 {
1650
1651     public:
1652
1653        //  ZZ methods...
1654
1655  };
1656
1657  class AAAAA: public virtual ZZ
1658 {
1659
1660     public:
1661
1662        //  AAAAA methods...
1663
1664  };
1665
1666  class BBBBB: public virtual AAAAA
1667 {
1668
1669     public:
1670
1671        //  BBBBB methods...
1672
1673  };
1674
1675  class CCCCC: public virtual BBBBB
1676 {
1677
1678     public:
1679
1680        //  CCCCC methods...
1681
1682  };
1683
1684  class DDDDD: public virtual CCCCC
1685 {
1686
1687     public:
1688
1689        //  DDDDD methods...
1690
1691  };
1692
1693  class EEEEE: public virtual DDDDD
1694 {
1695
1696     public:
1697
1698        //  EEEEE methods...
1699
1700  };
1701
1702  class FFFFF: public virtual EEEEE
1703 {
1704
1705     public:
1706
1707        //  FFFFF methods...
1708
1709  };
1710
1711  class GGGGG: public virtual FFFFF
1712 {
1713
1714     public:
1715
1716        //  GGGGG methods...
1717
1718  };
1719
1720  class HHHHH: public virtual GGGGG
1721 {
1722
1723     public:
1724
1725        //  HHHHH methods...
1726
1727  };
1728
1729  class IIIII: public virtual HHHHH
1730 {
1731
1732     public:
1733
1734        //  IIIII methods...
1735
1736  };
1737
1738  class JJJJJ: public virtual IIIII
1739 {
1740
1741     public:
1742
1743        //  JJJJJ methods...
1744
1745  };
1746
1747  class KKKKK: public virtual JJJJJ
1748 {
1749
1750     public:
1751
1752        //  KKKKK methods...
1753
1754  };
1755
1756  class LLLL: public virtual KKKKK
1757 {
1758
1759     public:
1760
1761        //  LLLL methods...
1762
1763  };
1764
1765  class MLLL: public virtual LLLL
1766 {
1767
1768     public:
1769
1770        //  MLLL methods...
1771
1772  };
1773
1774  class NLLL: public virtual MLLL
1775 {
1776
1777     public:
1778
1779        //  NLLL methods...
1780
1781  };
1782
1783  class OLLL: public virtual NLLL
1784 {
1785
1786     public:
1787
1788        //  OLLL methods...
1789
1790  };
1791
1792  class PPPP: public virtual OLLL
1793 {
1794
1795     public:
1796
1797        //  PPPP methods...
1798
1799  };
1800
1801  class QQQQ: public virtual PPPP
1802 {
1803
1804     public:
1805
1806        //  QQQQ methods...
1807
1808  };
1809
1810  class RRRR: public virtual QQQQ
1811 {
1812
1813     public:
1814
1815        //  RRRR methods...
1816
1817  };
1818
1819  class SSSS: public virtual RRRR
1820 {
1821
1822     public:
1823
1824        //  SSSS methods...
1825
1826  };
1827
1828  class TTTT: public virtual SSSS
1829 {
1830
1831     public:
1832
1833        //  TTTT methods...
1834
1835  };
1836
1837  class UUUU: public virtual TTTT
1838 {
1839
1840     public:
1841
1842        //  UUUU methods...
1843
1844  };
1845
1846  class VVVV: public virtual UUUU
1847 {
1848
1849     public:
1850
1851        //  VVVV methods...
1852
1853  };
1854
1855  class WWWW: public virtual VVVV
1856 {
1857
1858     public:
1859
1860        //  WWWW methods...
1861
1862  };
1863
1864  class XXXX: public virtual WWWW
1865 {
1866
1867     public:
1868
1869        //  XXXX methods...
1870
1871  };
1872
1873  class YY: public virtual XXXX
1874 {
1875
1876     public:
1877
1878        //  YY methods...
1879
1880  };
1881
1882  class ZZ: public virtual YY
1883 {
1884
1885     public:
1886
1887        //  ZZ methods...
1888
1889  };
1890
1891  class AAAAA: public virtual ZZ
1892 {
1893
1894     public:
1895
1896        //  AAAAA methods...
1897
1898  };
1899
1900  class BBBBB: public virtual AAAAA
1901 {
1902
1903     public:
1904
1905        //  BBBBB methods...
1906
1907  };
19
```

45

50

55

```

// C methods...

private:
5    Narrowable *create_shadow(ShadowMap *);

};

class D: public virtual A, public virtual B
10 {
public:
    // D methods...

15    Narrowable *create_shadow(ShadowMap *);

};

class Roman_ShadowMap: public virtual ShadowMap
20 {
public:
    virtual Narrowable *shadowRoman(Roman *);

25    virtual Narrowable *shadowA(A *);

    virtual Narrowable *shadowB(B *);

    virtual Narrowable *shadowC(C *);

30    virtual Narrowable *shadowD(D *);

};

```

Given the name of the root class, "Roman", the utility "autodefine" can mechanically generate the definition of the class "Roman_ShadowMap" and default method bodies. The details of the default "shadow T" methods are described below. Those skilled in these arts will recognize that "virtual inheritance" is used in this example as a matter of style, and it may not be necessary in all situations.

Making a specific shadow of a data structure

For any data structure to be shadowed, it must be built with the conventions described above. To permit a specific shadow to be made, the major task is to provide an implementation of the subtype of "ShadowMap" specific to the nodes in the data structure.

The shadow map subtype contains a collection of methods of the form:

45 Narrowable *shadowT(T *);

These methods will be implicitly invoked as a result of a client calling T'::shadow (t, map); for the first time. (Subsequent calls will return the same result, which will have been saved in the shadow map.) Thus it is the task of the method to construct a T' object from the T argument. This will typically involve shadowing the members of the T object and constructing a T' object from them.

50 In addition to providing an implementation of a shadow map with factory methods that actually create the nodes of the shadow data structure, a shadow method must be declared on each of the classes for nodes in the shadow data structure. More accurately, it must be declared on those classes which can be generated as the result of shadowing a shadowable node. (See T' ::shadow discussion above.)

If the class being shadowed is T, and the corresponding shadow class is T', then the shadow method is declared as:

```

class T
{
5    public:
    static T* shadow(T *t, TRoot_ShadowMap *map);
};

10

```

In the preferred embodiment, the implementation of this method is boiler-plate, and can be generated by the utility "autodefine". It calls map->lookup_or_create_shadow (t) and narrows the result to be of type T*. "Narrows" here refers to a C++ technology which is described in Appendix A. Narrowing is also known as "down-casting" in the C++ community.

15 Referring now to **Figure 13**, the process followed by a subsequent programmer 250 to make a shadow data structure from some pre-existing shadowable data structure is depicted. The programmer writes header files 254 to describe the objects that go to make up the shadow data structure. These header files may reference the header files (204 in **Figure 11**) supplied by the programmer of the original data structure. Since the shadow data structure is composed of objects, the programmer also writes code files 20 256 for the methods of these objects. These code files 256 will reference the header files supplied by the programmer of the original data structure (212 in **Figure 11**). The code for the "shadow factory" methods are also written 258. The header files 254 are combined with the header files 214 (in both **Figures 11** and 13) and passed through the utility "autodefine" 260 to generate the boilerplate code to produce the shadow data structure. The C++ code output by "autodefine" and the newly written code files 256 along with all of 25 the header files are compiled together 262 and the outputted object code 263 is combined with the object code supplied 220 (in both **Figures 11** and 13) and linked together to form a program 264. This program is then executed 266 in **Figure 13** and as shown in 270 in **Figure 14**. Referring now to **Figure 14**, during the program execution, a shadowable original data structure is created 274, a shadow data structure is created 276, and the shadowed data structure is executed 278. The subroutine to create the shadow data structure 30 276 in **Figure 14** is now described in more detail with reference to **Figure 15**. In **Figure 15**, the subroutine 290 when called 292 first creates a shadow map and associated shadow factory if it has not already done so 294. Then the subroutine 290 determines whether the original data structure node in question has been registered in the shadow map 296. If so 299 the subroutine returns the value registered (pointer) in the shadow map against the node in question 304, to the caller of the subroutine and exits 306. 35 If however, the original data structure node in question has not been registered in the shadow map 298, then the "create_shadow" method is called for the node, according to the node's type 300. This creates a shadow node and the result (pointer to the shadow node) is registered in the shadow map 302. This value then is returned to the caller of the subroutine 304 and the subroutine exits 306. The details of the "create_shadow" method (300 in **Figure 15**) is now described with reference to **Figure 16**. In **Figure 16**, 40 when the "create_shadow" method 310 is called 312, the implementation first determines the true type of the referenced node 314. A method is called on the shadow factory appropriate to the type of the referenced node object 316 to create a shadow node, and the newly created shadow node reference is returned to the caller 318 and the "create_shadow" method exits 320. The details of the method for the shadow factory (316 in **Figure 16**) are shown in **Figure 17**. In **Figure 17**, the shadow factory method 330, 45 when called 332, checks the shadowable node to see if there are any child nodes of the specified node which must be shadowed 334. If so, the subroutine is run to create shadow nodes for the child nodes of the original data structure 336. In so doing this step determines whether a loop would be caused and records the arguments for later processing. Finally the shadow factory implementation creates the shadow node from shadowed children and from other non-shadowable information from the shadowable node 338 and 50 exits 340.

Example: Shadow methods and ways of generating shadow nodes

Continuing the example above, involving classes A, B, C, and D, that inherit from "Roman", suppose we 55 wish to make a shadow involving nodes *Alpha*, *Beta*, *Gamma*, and *Delta*, that have a common root node, "Greek". Then, picking "Alpha" as an example, its "shadow" method might be defined as:

```

class Alpha
{
5    public:
        // other Alpha methods...
10    static Alpha *shadow(A *a, Roman_ShadowMap *map);
    };

```

Now a definition of a factory class is needed, such as "GreekFromRoman_ShadowMap" that is an implementation of the class "Roman_ShadowMap". Note that that class merely declared abstract (or default) "shadow T" methods for the various sub types of the root node, "Roman". It did not embody any knowledge of the types of the shadow nodes to be created; that knowledge will be embodied in the implementation of the methods of "GreekFromRoman_ShadowMap".

This is what the definition of the class "GreekFromRoman_ShadowMap" might look like.

```

20    class GreekFromRoman_ShadowMap:
        public virtual Roman_ShadowMap
    {
25        public:
            // Roman_ShadowMap methods defined by this class:
            Narrowable *shadowA(A *);
            Narrowable *shadowB(B *);
            Narrowable *shadowC(C *);
            Narrowable *shadowD(D *);
    };

```

If the shadow map being inherited has only pure virtual methods, then all of those methods will need to be implemented here. If there are any default implementations, then it may not be necessary to implement all of the methods here. Default factory methods for a shadow map are described below.

There are several paradigms that can be used, and the examples that follow should be considered merely illustrative of the techniques that can be used. Which paradigm is appropriate in any particular situation will be a matter of personal taste and considerations of how to group the shadowing code: all together, or with the code for the nodes to be shadowed.

Shadowing instance data within the factory method for a shadow node

Suppose we wish to shadow an A object into an Alpha object, and that an A object has two members, of type B* and C* that are both to be shadowed into the Alpha object. Then, assuming suitable access functions and constructors, the factory method could be written in the following way:

```

50    Narrowable *GreekFromRoman_ShadowMap::shadowA (A *a)
    {

```

```

Beta *beta = Beta::shadow(a->get_b(), this);
Gamma *gamma = Gamma::shadow(a->get_c(), this);
return new Alpha (beta, gamma);

5

```

The method shadows the child members of A, recursively invoking the shadow machinery for the children and calling a constructor on the result. Not all children need be shadowed if there is no need for them in the shadow node. Some members may not be shadowable: for example, basic types, enumerated 10 values, strings and so on. These members can either be copied across to the shadow node directly, or a back pointer to the original node can be stored in the shadow, so that the shadow can get at these members by following the back pointer. The decision of which way to store them should be made by taking considerations of space and access time into account.

Just as not all members of the original node may be shadowed, so not all members of the shadow node 15 may be the result of shadowing members from the original node. Storing a back pointer is a very simple example. Alternatively, some members of the shadow node may be the result of computing and caching a complex expression involving one or more members of the original node.

Shadowing instance data within a constructor for a shadow node

20 Instead of shadowing the children in the factory method, the shadowing can be done in the constructor for the shadow node itself:

```

25   Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
    return new Alpha (a, this);
};

30   Alpha::Alpha (A *a, Roman_ShadowMap *map)
      : beta(Beta::shadow(a->get_b(), map)),
      gamma(Gamma::shadow(a->get_c(), map()))
35   { };

```

40 Performing the shadowing in the constructor keeps the conversion code close to the rest of the code for Alpha, whereas performing the shadowing in the factory, groups all of the shadowing code (for all classes) together.

Shadowing instance data within a static method of the shadow node

45 Having the factory method call the constructor for the shadow node implies that the factory knows exactly what the shadow type really will be, as indicated by the calls of "new Alpha (...)". Another solution is to use a static create method on the Alpha class. This keeps the factory code simple, without binding in knowledge of a concrete type.

50

55

```

Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
    5
    return Alpha::create(a, this);
}
Alpha *Alpha::create(A *a, Roman_ShadowMap *map)
{
    10
    Beta *beta = Beta::shadow(a->get_b(), map);
    Gamma *gamma = Gamma::shadow(a->get_c(), map);
    return new Alpha(beta, gamma);
    15
}

```

Example: Generating different types of shadow node

20 There was an example above concerning a compiler parse tree node of type "PT_BinaryExpr" that is to be shadowed into one of a number of different types, depending on the binary operation. Here is how that might be implemented, using a static create method in the shadow class.

```

25
    CG_Expr *CGBinaryExpr::create(PT_BinaryExpr *be,
                                    PT_ShadowMap *map)
{
    30
    CG_Expr *left = CGExpr::shadow(be->get_left(), map);

    35
    CG_Expr *right = CGExpr::shadow(be->get_right(), map);
    switch (be->get_op()) {
        40
        case Op::Add:
            return new CG_AddExpr(left, right);
        case Op::Sub:
            return new CG_SubExpr(left, right);
        45
        case Op::Mult:
            return new CG_MultExpr(left, right);
        case Op::Div:
            return new CG_DivExpr(left, right);
        50
        default:
            should_not_happen();
    }
    55
};

```

Note that the body of this method could also be put into the factory method `shadowPT_BinaryExpr`. The choice of where to locate the code is mostly one of style, although by making it a method on `PT_BinaryExpr`, it is possible that a more sophisticated example would be able to make use of other methods defined in the class which might only be privately accessible.

5

Lazy shadows for cyclic data

The various examples of making shadows shown above also used the primitive operation "`T'::shadow`". As was noted in the explanation above, this will likely cause recursive calls on shadow for all the children of 10 the node being shadowed. The corollary is that when shadowing a node, the transitive closure of all children of that node will also be shadowed. This will work provided the data structure is acyclic. If the data structure is cyclic, then the recursive processing scheme will not terminate, and will eventually run out of memory.

"Lazy shadows" provide a way around these problems. They should be used if the data has (or might have) cycles; they can also be used if it is not necessary or desirable to shadow all of a data structure up 15 front.

If "`T`" is a type of node to be shadowed and "`T'`" is the type of the shadow node for `T`, then "`LazyShadow(T',T)`" is a template for a smart pointer that lazily evaluates "`T' :: shadow`". The smart pointer is initialized with the two arguments for "`T' :: shadow`:" a "`T'`" pointer and a "`ShadowMap*`". It can be assigned to a `T'` pointer, and can be transparently dereferenced as a `T'` pointer; both of these operations 20 will cause the shadow to be evaluated. Consider the following code, written without using lazy shadows.

```

Narrowable *GreekFromRoman_ShadowMap::shadowA(A *a)
{
    25
    return new Alpha(a, this);
}

class Alpha
{
    30
    public:
        Alpha(A *a, Roman_ShadowMap *map);
        void print();
    private:
        Beta *beta;
        Gamma *gamma;
    };
    35

Alpha::Alpha(A *a, Roman_ShadowMap *map)
    :
    40
    beta(Beta::shadow(a->get_b0, map)),
    gamma(Gamma::shadow(a->get_c0, map))
};

void Alpha::print()
{
    45
    Beta *beta2 = beta;
    gamma->print();
}

```

If for some reason it becomes necessary or desirable to use lazy shadows for the child nodes of class `Alpha`, here is how the code might change.

```

50
class Alpha
{

```

55

```

public:
    Alpha::Alpha(A *a, Roman_ShadowMap *map);
    void print();
5     private:
        LazyShadow<Beta,B> beta;           // was Beta *beta;
        LazyShadow<Gamma,C> gamma;         // was Gamma *gamma;
    };

Alpha::Alpha(A *a, Roman_ShadowMap *map)
10    :beta(a->get_b(), map), // shadow call removed
        gamma(a->get_c(), map) // shadow call removed
    { }

```

15 The method "GreekFromRoman_ShadowMap::shadow A" remains unchanged. The implementation of "Alpha::print" also remains unchanged, because "beta" can be transparently converted to and used as a "Beta" pointer, just as "gamma" can be converted to a "Gamma" one. The first time "beta" is converted to a "Beta" pointer, the call of "Beta::shadow(a->get_b(), map)" will finally occur; however, subsequent uses of "beta" will bypass the call and will be almost as fast as using the evaluated pointer.

20 **Instantiating a shadow of a data structure)Or: What the end client actually needs to know.)**

Most of the apparatus needed to instantiate a shadow of a data structure in the preferred embodiment has been shown. Those skilled in these arts will recognize any numbers of alternative embodiments. However, in the preferred embodiment, additionally, to shadow a data structure a client needs the following:

25 a root pointer to the data structure to be shadowed, or to a part of the data structure to be shadowed the shadow map subtype for the appropriate shadow type space

The following is a typical code sequence used to instantiate a shadow:

```

30     A *a = ...
           // get pointer to data structure to be shadowed

           Roman_ShadowMap *map = new GreekFromRoman_ShadowMap;
           Alpha *alpha = Alpha::shadow(a, map);
35           // that's all there is to it

```

40 This works as follows: after getting a pointer to the node to be shadowed, the code creates an instance of the appropriate subtype of "ShadowMap", and then simply shadows the root pointer, using exactly the same method we have already seen when shadowing the child members of a node being shadowed. The result is that "alpha" is a root pointer to a shadow data structure as defined by the "GreekFromRoman_ShadowMap" object.

45 Once the node has been shadowed, it depends on the specification of "GreekFromRoman_ShadowMap" as to when the map can be deleted. If the "alpha" shadow node has been fully evaluated, containing no internal lazy shadows and with no other implicit dependency on the map, then the map can indeed be deleted. However, a client may choose to keep the map available to shadow other nodes, in which case some or all of the saved entries in the map may speed up subsequent shadow operations.

50 **Additional Information for the Preferred Embodiment**

The above disclosure has given an overview of the entire mechanism, and details of what is needed both to make a data structure shadowable and to actually make a shadow. This section describes the glue code needed to make the various elements of code come together in the preferred embodiment.

55 Following the execution of the two lines of example code from the previous page:

```

    Roman_ShadowMap *map = new GreekFromRoman_ShadowMap;
    Alpha *alpha = Alpha::shadow(a, map);
        // that's all there is to it

```

5

Creating a shadow map

10 The call of new "GreekFromRoman_ShadowMap" constructs an instance of an implementation of a subtype of "ShadowMap" that can be used to generate a 'greek' shadow of 'roman' nodes. The call will automatically initialize a table in the ShadowMap base class that will be used to cache the results of calling the factory methods provided by the subtype.

Creating a shadow node

15

The shadow node is created by invoking the shadow method on the required type. This is a class-static method whose implementation is of the form:

```

20     T' *T'::shadow(T *x, TRoot_ShadowMap *m)
{
    return T'::narrow(m->lookup_or_create_shadow(x));
}

```

25 The map is interrogated for the shadow of "x" by the call of "m->lookup_or_create_shadow(x)".

The result of "lookup_or_create_shadow" is of type "Narrowable", since the map cannot know the type that will actually be required. The result is therefore narrowed to the correct type by calling "T'::narrow". Assuming the shadow has been created correctly, this will give the required result. Otherwise, zero will be returned.

30 Ignoring some minor complexity to deal with detecting cycles while creating shadows, the call of "lookup_or_create_shadow" checks to see if a shadow node is already registered for the argument, and if not, it takes steps to create the shadow node. Somewhat simplified, the code looks like the following:

```

35     Narrowable *ShadowMap::lookup_or_create_shadow(Shadowable *sp)
{
    Narrowable *np;
    if (sp == 0) return 0;
    if (!tbl->get_shadow(sp, np)) {
        np = sp->create_shadow(this);
        tbl->set_shadow(sp, np);
    }
    return np;
}

```

The following notes are of interest in further explaining the above:

tb1 This represents the table that contains the results of earlier calls to lookup_or_create_shadow.

50 tb1->get_shadow(sp, np)

This looks up the shadow of "sp", sets "np" and returns TRUE if it is found, and otherwise leaves "np" alone and returns FALSE.

tb1->set_shadow(sp, np)

This records "np" as the shadow of "sp".

55 The major line that needs explaining is this one:

np = sp->create_shadow(this);

This is the means by which the shadow mechanism correctly creates the appropriate type of shadow node, independently of the perceived type of the object. The client may have a "B*" pointer that in reality

points to a "D" object. Nevertheless, the client may well invoke "Beta::shadow(b, map)" and will expect to receive a Beta object even though the real type of the object being shadowed is a D and the real type of the shadow node must be a Delta.

5 "create_shadow" is a virtual method of the node being shadowed. It is implemented for each type of object being shadowed, and looks like this:

```

10    Narrowable *D::create_shadow(ShadowMap *map)
    {
        Roman_ShadowMap *m = Roman_ShadowMap::narrow(map);
        return m->shadowD(this, map);
    }

```

15 Note that the argument is only of type "ShadowMap*" and not of type "Roman_ShadowMap*". This is a regrettable consequence of "create_shadow" being inherited from "Shadowable", which only knows about "ShadowMap".

When "sp->create_shadow(this)" is invoked, the message will be received by the true type of the object, in its implementation of "create_shadow." This knowledge of the true type is passed on to the map by invoking a method specific to the true type. This is the purpose of the call:

20 m->shadowD(this)

Since this is a call to shadow a "Roman" node (D), this is a method on the "Roman_ShadowMap", which is the reason for the "narrow" call in the previous line of code. Note that map is not narrowed to the actual implementation type of the shadow map, since that is neither known nor necessary; it is merely narrowed to the type that defines the abstract methods to shadow the various types.

25 Typically, the call of "m->shadowD(this)" will be implemented by a subtype of "Roman_ShadowMap" that provides the knowledge of how to shadow a D node into the appropriate shadow type. In our running example, the call would be handled by "GreekFromRoman_ShadowMap::shadowD", of which we have already seen the like. (See the discussion on the implementation of shadow A above.)

30 The result of the shadow call on the map is of type "Narrowable". The result is passed back to "ShadowMap::lookup_or_create_shadow", which saves the value for future use, and returns it, still as a "Narrowable", to the original call of shadow on the target type. This attempts to narrow to the appropriate type the value that has been passed back. If the narrow call succeeds, the client has the value that was wanted; if not, then zero is returned, which should be interpreted as meaning that either the item being shadowed was itself zero, or that the item cannot be shadowed to the requested type. If it is important to 35 distinguish these cases, the client can easily check if the item being shadowed is zero or not.

35 It is worth checking what happens in the face of inheritance and widened pointers. Assume the client has in his hand a B* pointer to a D object. As a B* pointer, it is to be expected that the shadow will be a Beta* pointer. Therefore, "Beta::shadow(b, map)" will be called. Assuming this is the first shadow call on "b", "ShadowMap::lookup_or_create_shadow" will invoke "create_shadow" on "b", which will be handled by the true type D, by way of the virtual function dispatch. This will call the factory to shadowD. Thus the shadow node will be created with the correct shadow type. The pointer to the shadow node (e.g. a Delta*) will be passed back as a "Narrowable", and narrowed to Beta * in the code for "Beta::shadow". Thus the mapping between node type and shadow node type is maintained for both the true and perceived type of the objects involved.

45

Who does what

50 There are many different parts to the shadow system disclosed herein, with the code for the various parts being the responsibility of various different authors in the preferred embodiment. There are at least three authors involved: the author of the standard shadow code, the author of the data structure being shadowed and the author of the shadow data structure itself.

In addition, some of the code can be mechanically generated by a special utility called autodefine which is described in Appendix B. Also described in Appendix A is additional explanatory information on the "Narrow" implementation used in the preferred embodiment.

55 Referring to **Figure 19** a table describes the various components of the shadow system and identifies the responsibility for the various parts in the preferred embodiment.

Default factory methods on a shadow map

The disclosure of the preferred embodiment above described the actions necessary make a data structure shadowable, and introduced the need for a subtype of "ShadowMap" that declared a "shadowT" method for each class "T" in the data structure that needed to be shadowed.

The disclosure above indicated that these methods could be pure virtual methods, and that the class could be automatically generated by "autodefine". The following illustrates how default methods on the "ShadowMap" subtype may be used to advantage.

10 The examples so far have suggested that each type in the data structure has one or more corresponding types in the shadow data structure: examples of both 1-1 and 1-many mappings were illustrated. However, many-1 mappings (and even many-many mappings) are also possible.

Consider the inheritance hierarchy 370 shown in Figure 20. Nodes A 373, B 376, and C 378 inherit from "Roman" 372; and nodes C1 380, C2 382 and C3 384 inherit from C 378.

For a particular shadow of a data structure built from these data types, it may be the case that we do not need to shadow all of the information in types C1 380, C2 382 and C3 384 : it might be sufficient just to shadow the properties they have as C 378 objects. In other words, rather than shadowing C1 to Gamma1, C2 to Gamma2, and C3 to Gamma3, we wish to shadow all C nodes to Gamma nodes.

Since we wish to shadow nodes of these types to Gamma nodes, it is appropriate to call "Gamma: shadow (C*, Roman_ShadowMap)". Assuming this is the first shadow call on the C * argument, this will call through the virtual create_shadow method, and that call will be received by the true type of the object: C1, C2 or C3. Whichever type receives the call will pass it on to the appropriate method on the factory: shadowC1, shadowC2 or shadowC3.

If the methods on the "Roman_ShadowMap" are all pure virtual methods, then they must all be separately implemented by the actual map used for the specific shadow. However, if the methods on "Roman_ShadowMap" have a default implementation, then they do not need to be overridden in the actual map used. In this case, what is wanted is for shadowC1, shadowC2 and shadowC3 all to delegate to shadowC. This means that in the actual map to be used, only shadowC need be provided, and this can do what is necessary to shadow any C objects, whether they be C, C1, C2 or C3 objects.

Based on this rationale, the following rule is implemented by autodefine in the present embodiment when generating the "TRoot_ShadowMap" for a collection of types inheriting from TRoot. The default implementation for each shadowT method is to delegate to the immediate parent type, in the case of single inheritance, or to the leftmost immediate parent, in the case of multiple inheritance. ("Autodefine" in the preferred embodiment also has an option to make the methods on the "TRoot_ShadowMap" class be pure virtual methods. It should also be noted that "autodefine", in the preferred embodiment, does not notice inherited template types.) The one exception is the shadowTRoot method, to which all methods will eventually delegate if not overridden, and this method will abort if invoked. It is up to the client to ensure that this abort does not get called: either by overriding it, or by ensuring that no instances of TRoot are allocated and that all subtypes of TRoot have overridden appropriate shadowT methods in the map.

40 **Shadows of shadows**

If some data structure is shadowed to yield a new data structure, there is no reason why that new data structure should not itself be shadowable. All that is required is that the base type of the nodes in the new data structure should inherit from `Shadowable`, and that the nodes should implement `create_shadow`.

45 A sequence of shadows could be used (for example) to represent the transformations involved in a compilation pipeline, going from syntax tree to machine code instructions. In this case, the job of fabricating the shadow nodes at each stage would probably be non-trivial, but there is no inherent reason why this could not be done.

A variant of this is that a shadow might provide methods that, when invoked, cause a different data structure to be created, and this resultant data structure might itself be shadowable.

Passing extra parameters to the shadow call

When shadowing a node, it is sometimes useful to be able to pass additional information to the call that will fabricate the shadow node. This may be information needed globally by the whole shadow (such as a window handle or error stream) or it may be needed by a specific type and a specific shadow call.

If this is necessary, such information can be stored as instance data of the actual implementation of the `ShadowMap`. This means that the information will be available in any call of `shadowT` that may subse-

quently be invoked.

Information that is global to the shadowing process can be stored as instance data in the map when the map is constructed. Information that is needed on a per-node basis requires a little more care. The easy solution is to provide access methods on the factory that directly allow the extra instance data to be accessed and modified throughout the shadowing process. However, if this technique is used, there is a serious potential for an undetected programmer error. Consider the following scenario:

Inside some call of "Tx' :: shadow (Tx *X, TRoot_ShadowMap "map)"

- set instance data "d" on the map: "map->set_data(d)"
- call "Ty' ::shadow(y, map)" to shadow some child "y"

The problem is that there is no inherent guarantee that the final call of "Ty'::shadow (y, map)" will see the data "d". If by some programmer error the shadow call has already been executed for the value y, then an entry will exist in the map and subsequent calls will return that value, and will not go through the factory method and pick up the data. In addition, it is clumsy for the client to have to call two methods: one to set the data and another to shadow the desired node. What is required is a slightly different mechanism that is easier to use and which provides better semantic guarantees.

In the preferred embodiment, the solution is to provide methods on the map that set the instance data and shadow the node in one single operation. Since the reason for these methods is to have extra shadow-specific parameters, these are methods defined directly on the implementation of the shadow map, and are not inherited at all. For example, the lines above could be replaced by a method defined as follows:

20

```

class TRoot' _from_TRoot_ShadowMap
{
25    public:
        Ty' *create_Ty_shadow(Ty *y, Data *d);
        // rest of class ...
30    private:
        Data *data_for_create_Ty_shadow;
    };

```

35

The implementation of

Ty' *create_Ty_shadow(Ty *y, Data *d)

can set the instance variable "data_for_create_Ty_shadow" and must then cause a shadow to be created. Whilst it can do that by explicitly calling

40

Ty' ::shadow(y, map)

there is another method on ShadowMap that is more suitable. Internally, "Ty' ::shadow(y, map)" calls Ty' :: narrow (map->lookup_or_create_shadow (y))

and as we have seen, this returns the value stored in the map if it already exists. The alternative is to replace the call of "Ty' :: shadow (y, map)" by the following

45

y' = Ty' :: narrow (map->create_shadow (y));

The call of "map->lookup_or_create_shadow(y)" has been replaced by a call to the sibling method "map->create_shadow(y)", which still looks up y in the map, but this time to check that no value is already stored in there. It is a run-time error if a value is found. Assuming no entry is found, the code will then internally call

50

y->create_shadow (map)

which, as before, will call back into one of the shadow T methods on the map. These all have access to the data d and can fabricate the shadow node correctly, based on the type of the node and the contextual information passed in from the client.

55

The name "create_Ty_shadow" is the choice of the author of the shadow map implementation class, in this case "TRoot' _from_TRoot_ShadowMap". It is specific to the implementation class and not inherited at all. It is suggested that it not be an overloaded name in case the implementation class is ever extended by implementation inheritance to shadow even more types.

Although the present invention has been described with reference to particular operating systems, compilers, program code mechanisms, and object and object reference definitions, it will be appreciated by one skilled in the art that the present invention may be implemented in any one of a number of variations within a given operating environment, or in different operating system or object system environments.

5 Similarly, particular computer client and server configurations or combinations illustrated are only representative of one of many such configurations of clients and servers and object and sub-object relationships which may use the present invention. Moreover, it will be understood that the figures are for illustration only and should not be taken as limitations on the invention. Some additional applications of the shadows techniques disclosed herein include the following:

10 A compiler's ASG can be shadowed with a data structure that more closely matches the requirement of a code generator. (For example, additional methods may be added to nodes to translate the program fragment represented by the node to machine code.);
 A compiler's ASG can be shadowed with a data structure that provides browsing capabilities over the program;

15 A compiler's ASG can be shadowed with a data structure that enables the ASG to be edited. The shadow nodes would contain some textual rendition of the nodes, the text could be edited, and then parsed back to an ASG;
 A compiler's ASG can be shadowed with a data structure that can be used as the input to an interpreter, to execute the program;

20 The schematics for the design of an integrated circuit can be represented as a connected collection of typed objects. Such data structures could be made shadowable to permit an extensible collection of tools to manipulate the data structure. For example, such tools as various design rule checkers for different manufacturing processes could be used. As another example the data structure may be shadowed into the form of data structure representing the information needed for a particular foundry to manufacture the chip;

25 Today, word processors manipulate complex structured documents. By making the document structure shadowable, various tools can be added to process the document, without building these tools into the one word processing program. Such tools might be style checkers, spelling checkers, cross-referencing programs, index construction, and converting to formats for other word processing programs.

These possible uses of the Shadow technique are not intended to limit in any way the possible uses of the Shadow functions as disclosed herein, but merely represent some examples which those skilled in these arts will recognize as merely exemplary.

Claims

35 1. A computer implemented method comprising the steps of:
 augmenting first header files to add a method to each type of object to be shadowed, said method operative to create a shadow of an object to be shadowed;
 scanning said first header files to look for a declaration of a standard method, and if said standard method is found, to generate a corresponding standard definition of said declaration;
 40 collecting said first header files for later use;
 compiling first code files and said first header files to produce a first object code corresponding to said first code files and said first header files; and
 combining said first header files collected for later use with said first object code, said combination representing a mechanism to process a shadowable data structure.

45 2. The computer implemented method described in claim 1 comprising the additional steps of:
 writing second header files to describe objects which will make up a shadow data structure;
 writing second code files for methods of said objects which will make up a shadow data structure;
 combining said second header files, said second code files and said first header files into a first combination of files;
 50 compiling said first combination of files to produce a second object code corresponding to said first combination of files;
 linking said first object code and said second object code to form a first program; and
 executing said first program.

55 3. The computer implemented method described in claim 2 wherein said step of executing said first program comprises the additional steps of:
 executing a subprogram to create a shadow data structure; and

processing said shadow data structure.

4. A computer implemented method comprising the steps of:
 - providing a first data-structure;
 - constructing a second data-structure which is related to said first data-structure by calling a program which is operative to create an element for said second data-structure and to create and store a pointer mechanism, said pointer mechanism being operative to identify said element; and
 - using said program to relate an element in said second data-structure to a corresponding element in said first data-structure by means of said pointer mechanism, a first of said pointer mechanisms pointing to an element in said first data-structure and a second of said pointer mechanisms pointing to an element in said second data-structure.
5. The computer implemented method described in claim 4 further comprising a step of using said second data-structure as an executable variation of said first data-structure.
15. 6. The computer implemented method described in claim 4 wherein said first data structure and said second data-structure are connected collections of object-oriented programming objects.
20. 7. The computer implemented method described in claim 6 wherein said element in said second data-structure is called a shadow node which is an object-oriented programming object, and said element in said first data-structure is a node which is an object-oriented programming object.
25. 8. The computer implemented method described in claim 4 wherein said step of constructing a second data-structure which is related to said first data-structure by calling a shadow-map comprises the steps of calling upon a factory method and caching results from said call to said factory method in said shadow-map.
30. 9. The computer implemented method described in claim 7 wherein said shadow-map is represented by a first object-oriented programming object which has a first method, said first method operative to look-up a shadow node in said shadow-map to determine whether an entry exists which points to said shadow node, and if said entry which points to said shadow node does not exist, said first method will create a new shadow node.
35. 10. The computer implemented method described in claim 9 comprising an additional step of, said first method, upon determining that an entry exists which points to said shadow node, returning said pointer to said shadow node to an originator of a look-up call.
40. 11. The computer implemented method described in claim 9 comprising an additional step of, said first method, upon creating a new shadow node, also creating a new entry in said shadow-map pointing to said new shadow node, and said first method returning said new pointer to said new shadow node to an originator of a look-up call.
45. 12. The computer implemented method described in claim 7 comprising the additional steps of making a particular node in said first data-structure shadowable by:
 - implementing in said particular node of said first data-structure, a create_shadow method, wherein said create_shadow method calls a second method on an object-oriented programming object which represents said shadow-map, said second method operative to fabricate a shadow node by passing in an object type of said particular node; and
 - providing an object-oriented programming object which defines a method for each node type to be shadowed.
50. 13. The computer implemented method described in claim 12 wherein said create_shadow method is a virtual method.
55. 14. The computer implemented method described in claim 4 wherein said first data structure is shadowable.

15. A computer implemented method for creating a shadow data-structure of an existing data-structure, said method comprising the steps of:

providing a shadowable first data-structure;

creating a shadow node for each node of said shadowable first data-structure by calling a shadow-map which is operative to create and store a pair of pointers, each of said pointers being operative to identify an element in a data-structure; and

relating each of said shadow nodes in said second data-structure to a corresponding node in said shadowable first data-structure by means of said pair of pointers in said shadow-map.

10 16. The computer implemented method for creating a shadow data-structure of an existing data-structure described in claim 15 comprising the additional step of :

using said shadow-map to relate an element in said second data-structure to a corresponding element in said first data-structure by means of said pair of pointers, a first of said pair of pointers pointing to an element in said first data-structure and a second of said pair of pointers pointing to an element in said second data-structure.

15 17. The computer implemented method for creating a shadow data-structure of an existing data-structure described in claim 15 wherein said shadow-map is represented by a first object-oriented programming object which has a first method, said first method operative to look-up a shadow node in said shadow-map to determine whether an entry exists which points to said shadow node, and if said entry which points to said shadow node does not exist, said first method will create a new shadow node.

20 18. The computer implemented method for creating a shadow data-structure of an existing data-structure described in claim 17 comprising an additional step of, said first method, upon creating a new shadow node, also creating a new entry in said shadow-map pointing to said new shadow node, and said first method returning said new pointer to said new shadow node to an originator of a look-up call.

25 19. The computer implemented method for creating a shadow data-structure of an existing data-structure described in claim 17 wherein said step of providing a shadowable data-structure comprises the step of making a particular node in said first data-structure shadowable by:

implementing in said particular node of said first data-structure, a create_shadow method, wherein said create_shadow method calls a second method on an object-oriented programming object which represents said shadow-map, said second method operative to fabricate a shadow node by passing in an object type of said particular node; and

providing an object-oriented programming object which defines a method for each node type to be shadowed.

30 20. The computer implemented method for creating a shadow data-structure of an existing data-structure described in claim 19 wherein said create_shadow method is a virtual method.

40 21. A System for extending the use of a data-structure comprising:

a computer;

a first data-structure coupled to said computer;

a device, coupled to said computer, which is operative to create a second data-structure, and to relate an element in said second data-structure to a corresponding element in said first data-structure by means of a pointing structure, a first element of said pointing structure pointing to an element in said first data-structure and a second of said pointing structure pointing to an element in said second data-structure; and

a computer code device, coupled to said second data-structure, for using said second data-structure as an executable variation of said first data-structure.

45 22. The System for dynamically extending the use of a data-structure described in claim 21 wherein said first data structure and said second data-structure are connected collections of object-oriented programming objects.

50 23. The System for dynamically extending the use of a data-structure described in claim 22 wherein said element in said second data-structure is called a shadow node which is an object-oriented programming object, and said element in said first data-structure is a node which is an object-oriented

programming object.

24. The System for dynamically extending the use of a data-structure described in claim 21 wherein said shadow-map device comprises a factory object coupled to said shadow-map device, and a caching device coupled to said shadow-map device for caching results from a call to said factory object.

5

25. The System for dynamically extending the use of a data-structure described in claim 23 wherein a first object-oriented programming object is coupled to said shadow-map device, said first object-oriented programming object comprising a first method, said first method operative to look-up a shadow node in said shadow-map device to determine whether an entry exists which points to said shadow node, and if said entry which points to said shadow node does not exist, said first method operative to create a new shadow node.

10

26. The System for dynamically extending the use of a data-structure described in claim 25 wherein said first method is also operative to create a new entry in said shadow-map device when creating a new shadow node, said new entry in said shadow-map device pointing to said new shadow node, and said first method is also operative to return said new pointer to said new shadow node to an originator of a look-up call.

15

27. The System for dynamically extending the use of a data-structure described in claim 23 further comprising a second program code device coupled to said computer, for making a particular node in said first data-structure shadowable, said second program code device having a third program code device for implementing in said particular node of said first data-structure, a create_shadow method, wherein said create_shadow method is operative to call a second method on an object-oriented programming object which represents said shadow-map device, said second method operative to fabricate a shadow node by passing in an object type of said particular node; and a fourth program code device, coupled to said third program code device, for providing an object-oriented programming object which defines a method for each node type to be shadowed.

20

28. The System for dynamically extending the use of a data-structure described in claim 27 wherein said create_shadow method is a virtual method.

30

29. A computer mechanism for projecting a first data-structure of object-oriented programming objects from one type space to another type space comprising:

35

- a first program code mechanism operative to convert said first data-structure of objects into a shadowable data-structure of objects;
- a device, coupled to said computer mechanism, which is operative to create a second data-structure of objects, and to relate an element in said second data-structure to a corresponding element in said first data-structure by means of a pointing structure, a first element of said pointing structure pointing to an element in said first data-structure and a second element of said pointing structure pointing to an element in said second data-structure; and

40

- a computer code device, coupled to said second data-structure, for using said second data-structure as an executable variation of said first data-structure.

45

30. A computer apparatus for constructing a shadow data-structure, said apparatus comprising:

- a query device for determining whether a specified node of said shadow data-structure exists;
- a create device, coupled to said query device, for creating said specified node of said shadow data-structure if said specified node does not already exist;
- a register device, coupled to said create device, for recording a value indicating that said specified node of said shadow data-structure has been created; and

50

- an answering device, coupled to said register device, for providing said value indicating that said specified node of said shadow data-structure has been created to a mechanism which issues a query as to the existence of said specified node.

55

31. The computer apparatus for constructing a shadow data-structure specified in claim 30 wherein said create device comprises:

- a checker device, coupled to said create device, for determining a true type of said specified node;
- a calling device, coupled to said checked device, for selecting a factory mechanism corresponding

to said true type of said specified node;
a selected factory mechanism, coupled to said calling device, for creating a shadow node; and
a notification device, coupled to said selected factory mechanism, for providing said shadow node
to said register device.

5

10

15

20

25

30

35

40

45

50

55

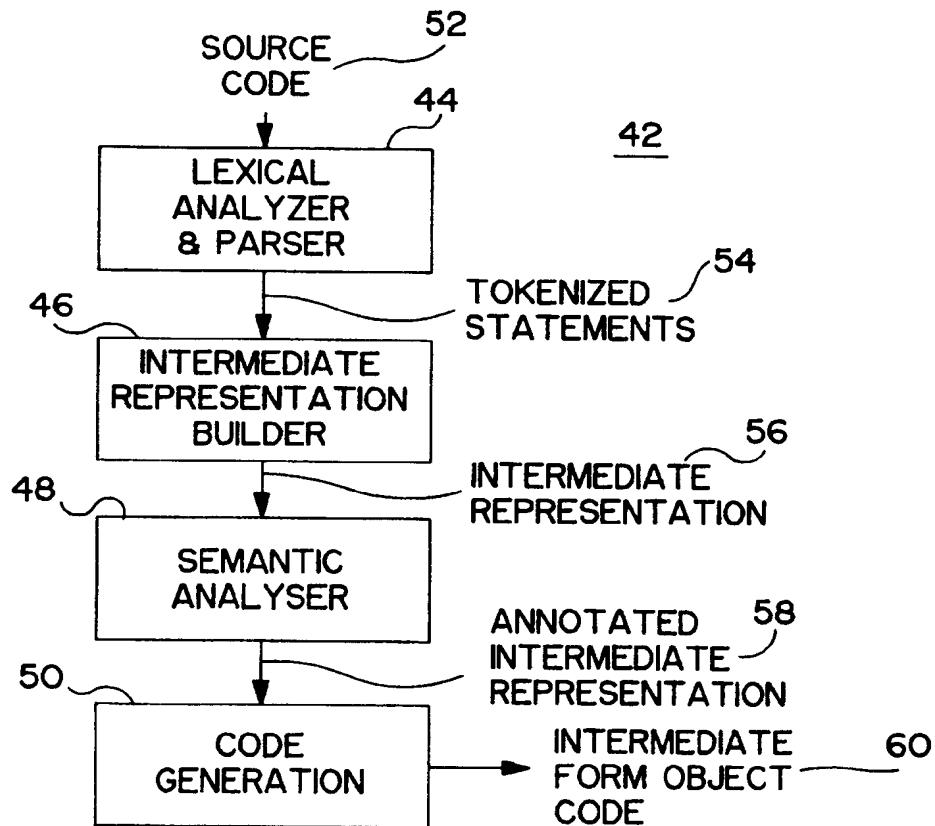
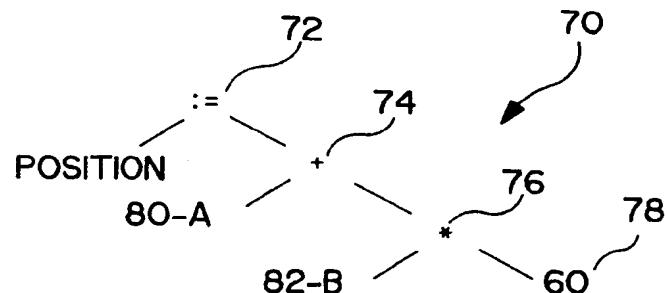


FIG. 1



SYNTAX TREE FOR "POSITION:= A + B * 60"

FIG. 2

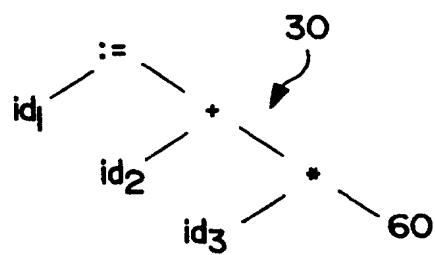


FIG. 3(a)

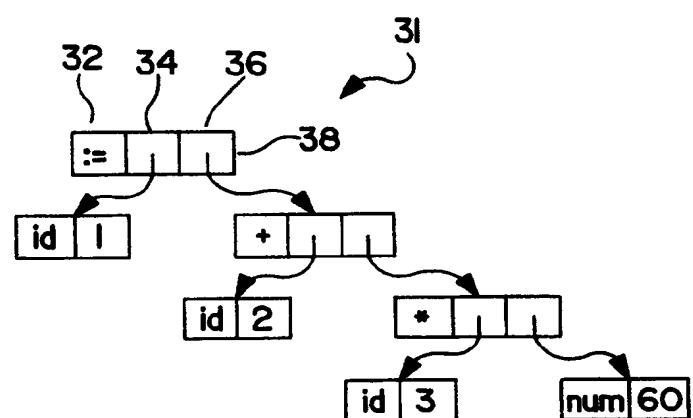
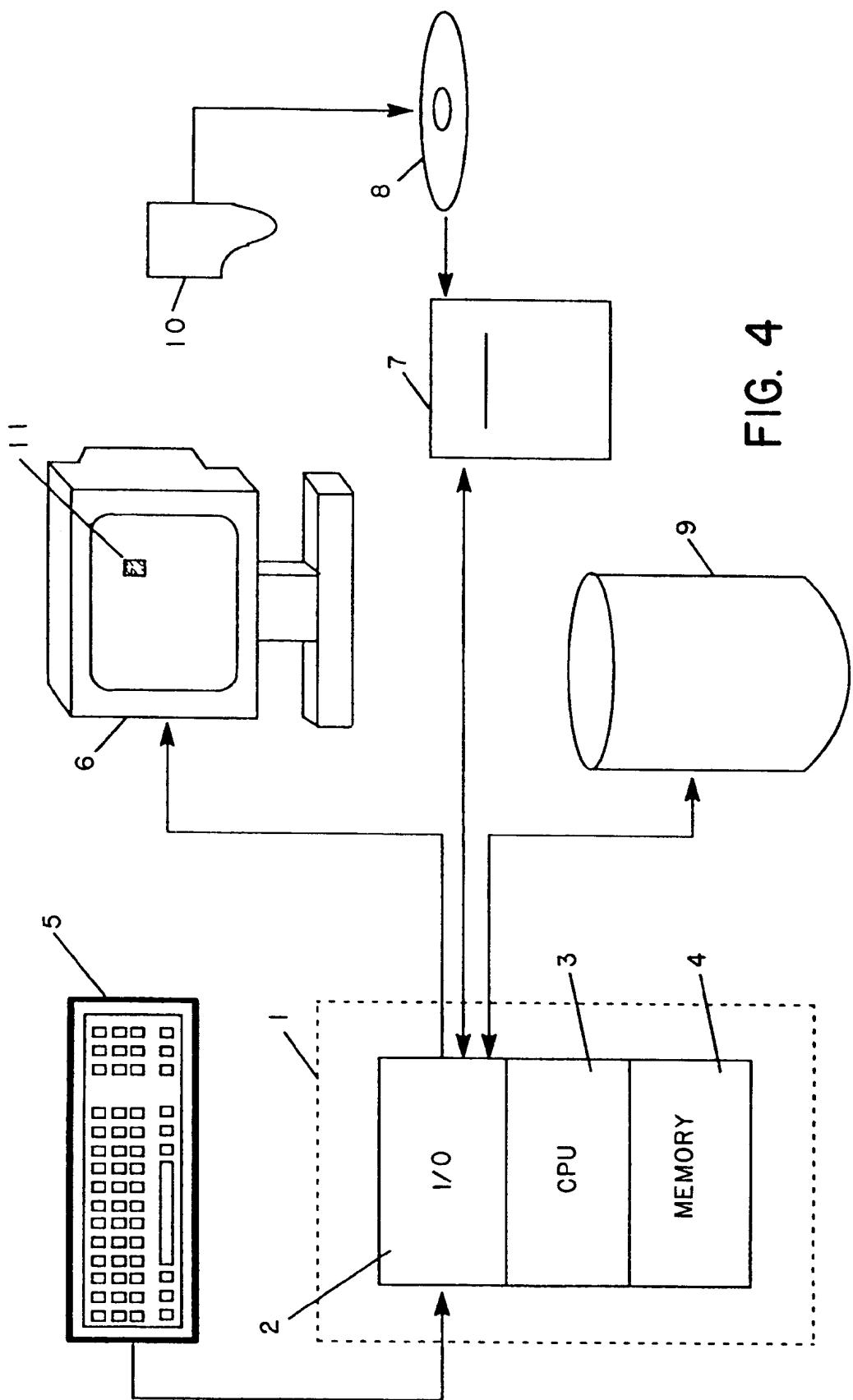


FIG. 3(b)



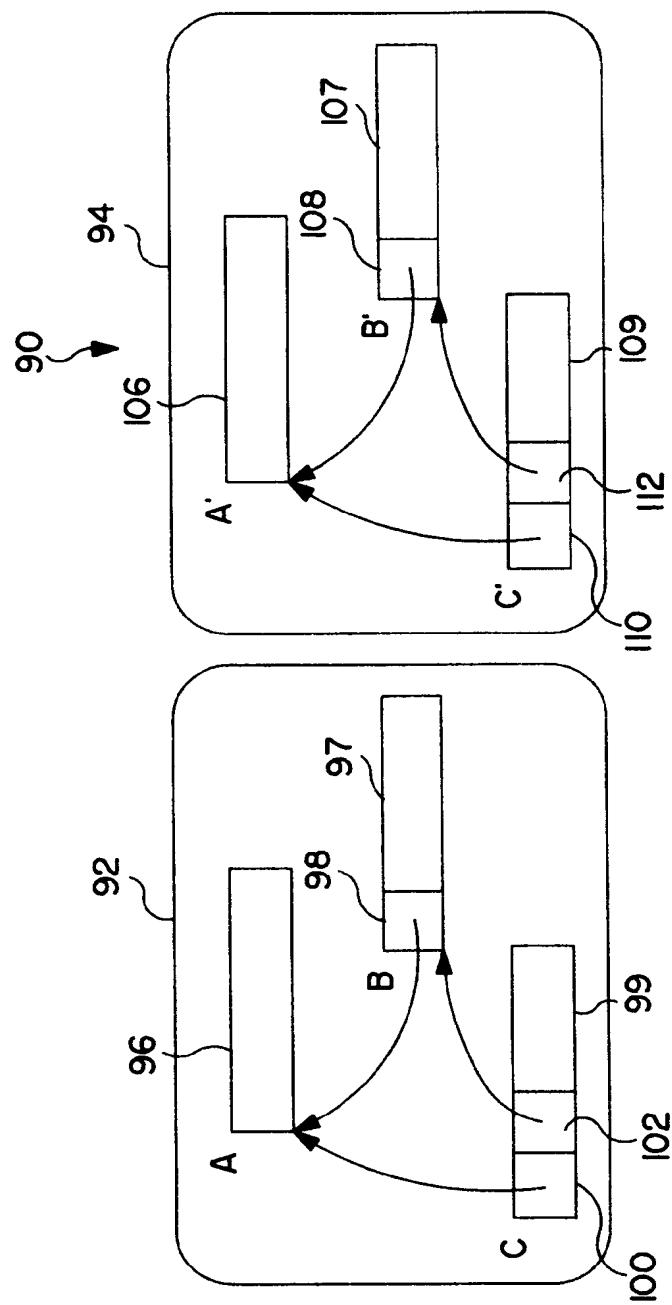


FIG. 5

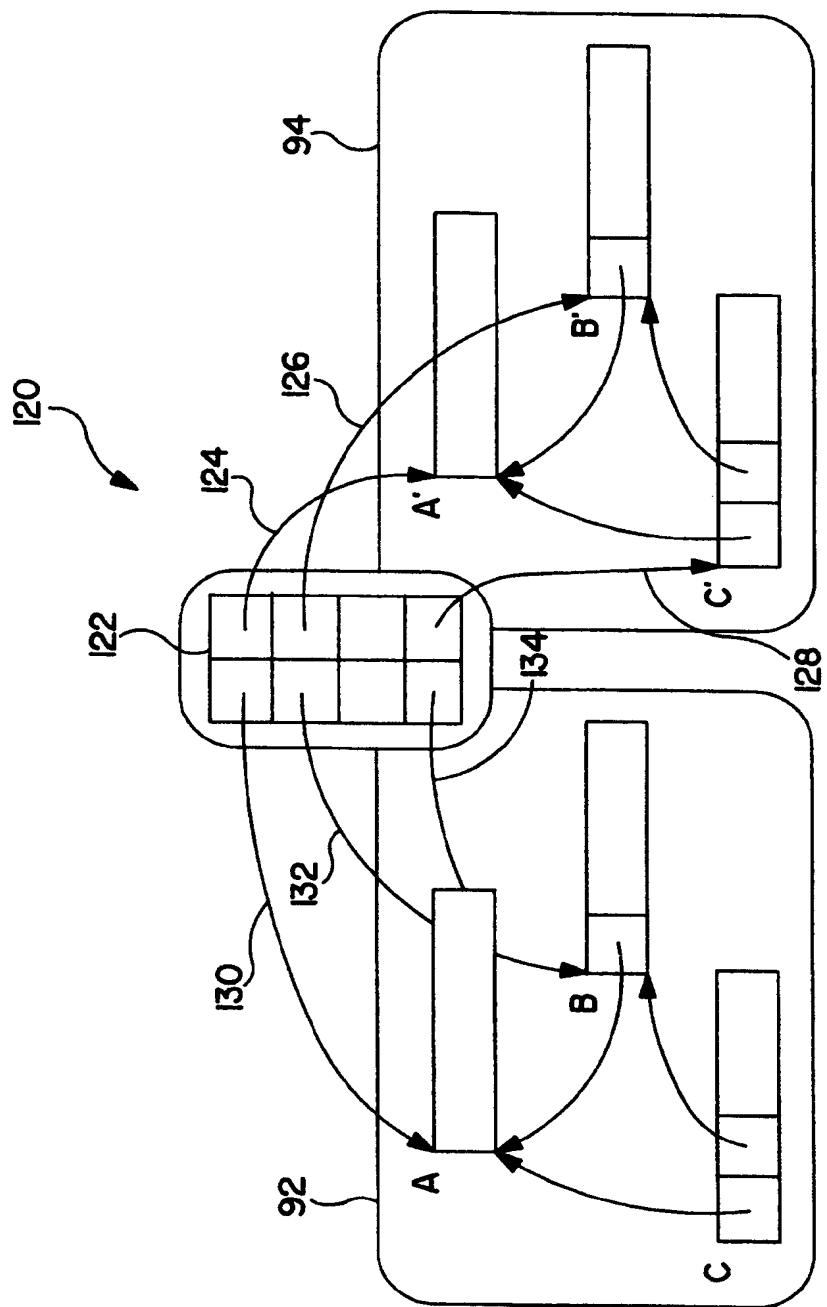


FIG. 6

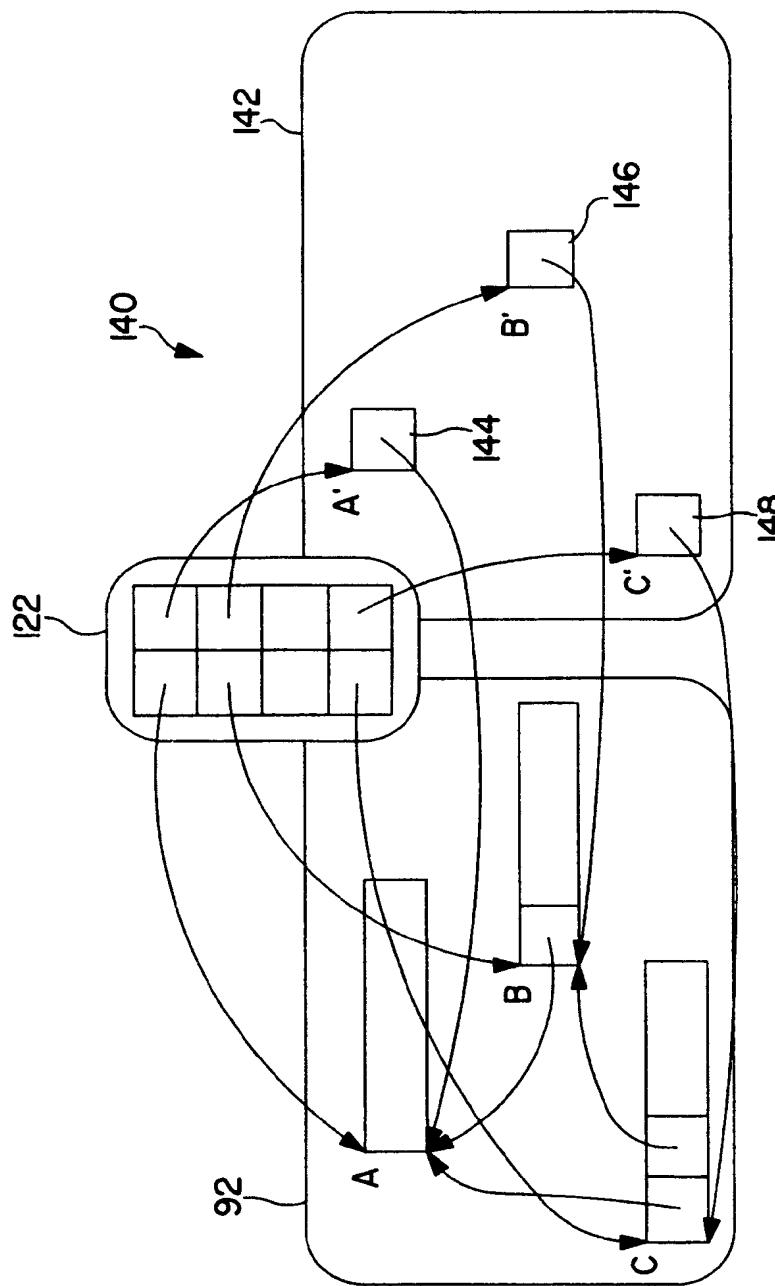


FIG. 7

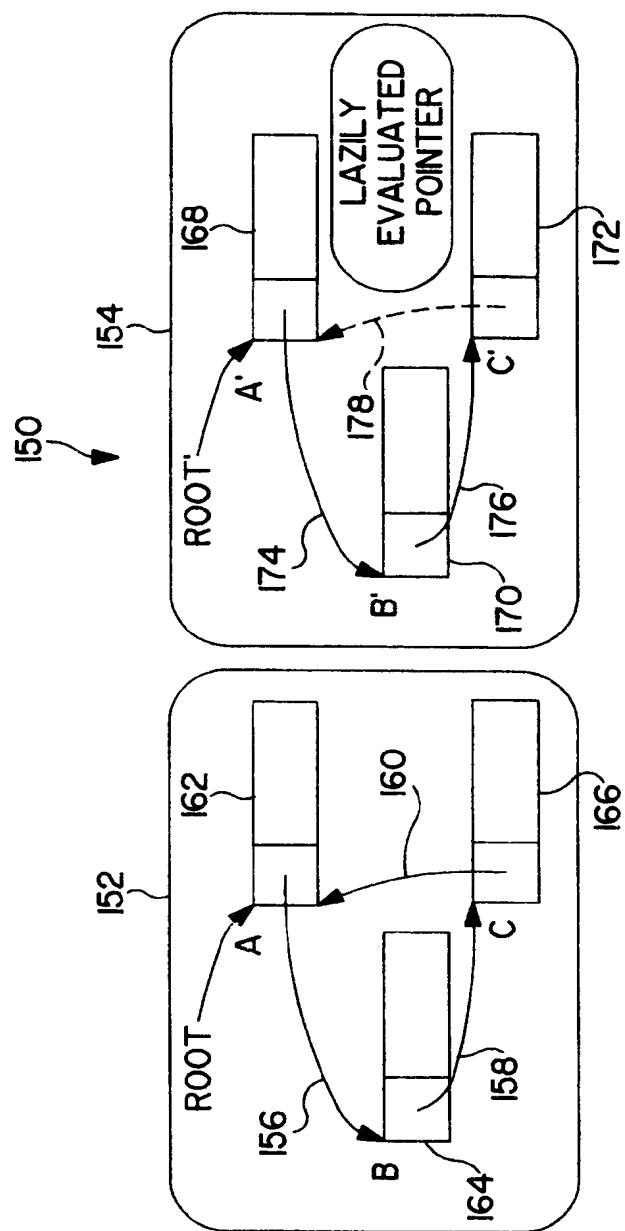
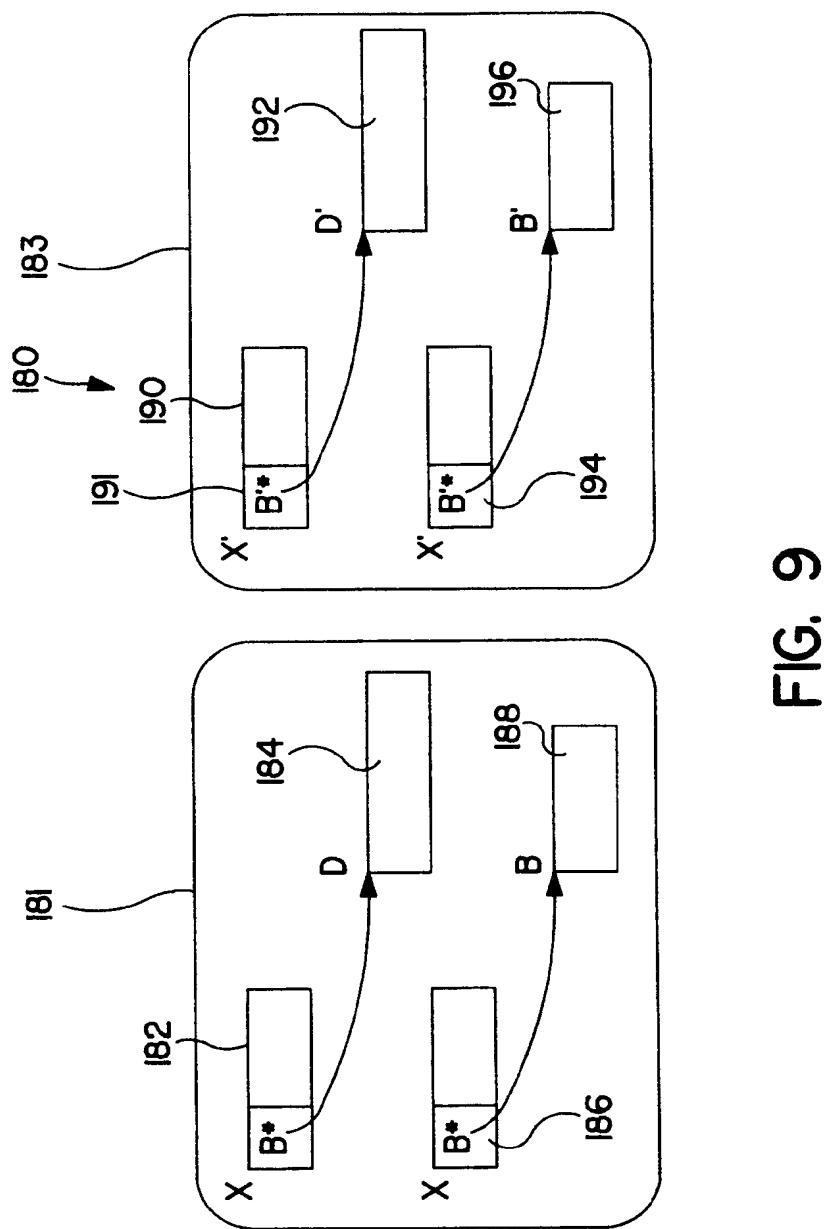


FIG. 8



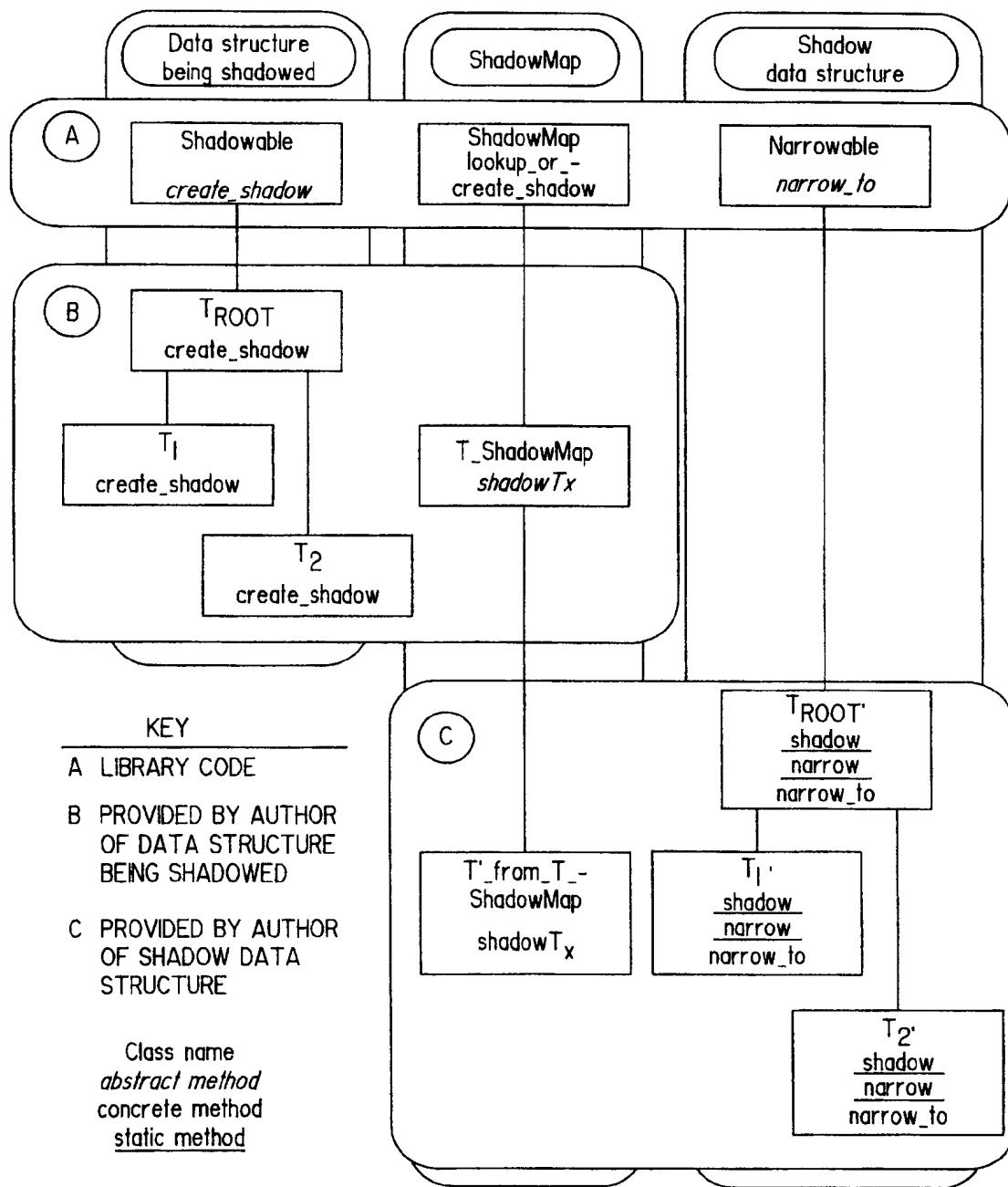


FIG. 10

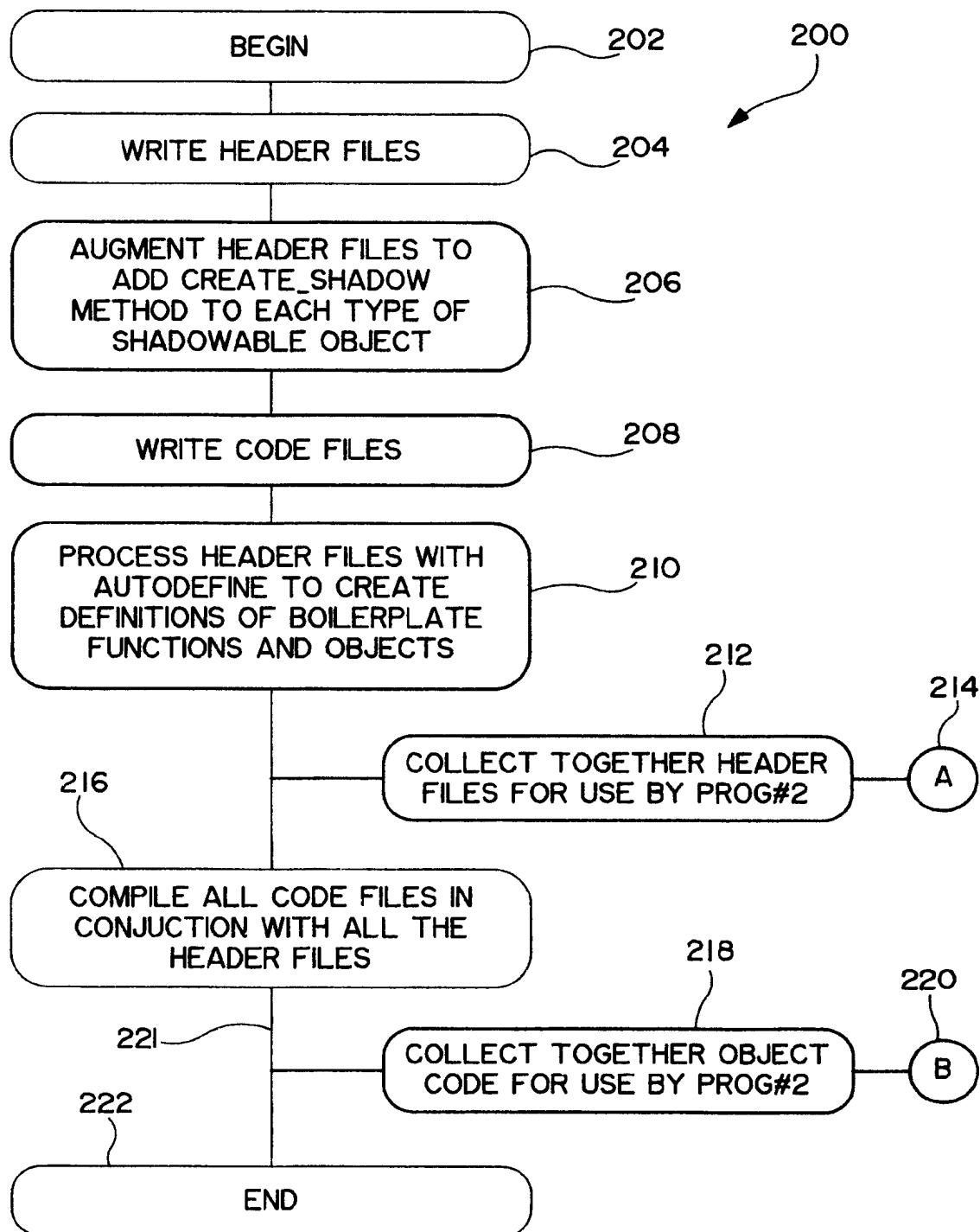


FIG. 11

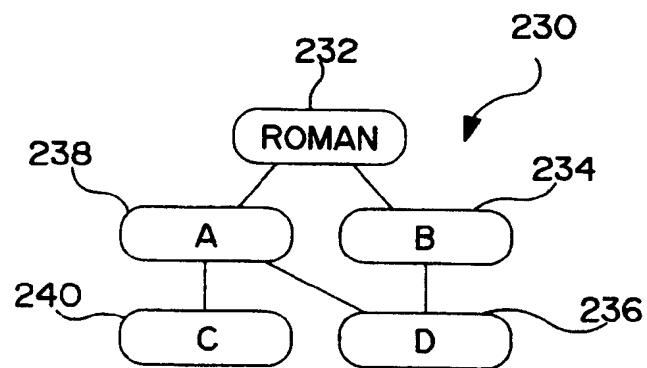


FIG. 12

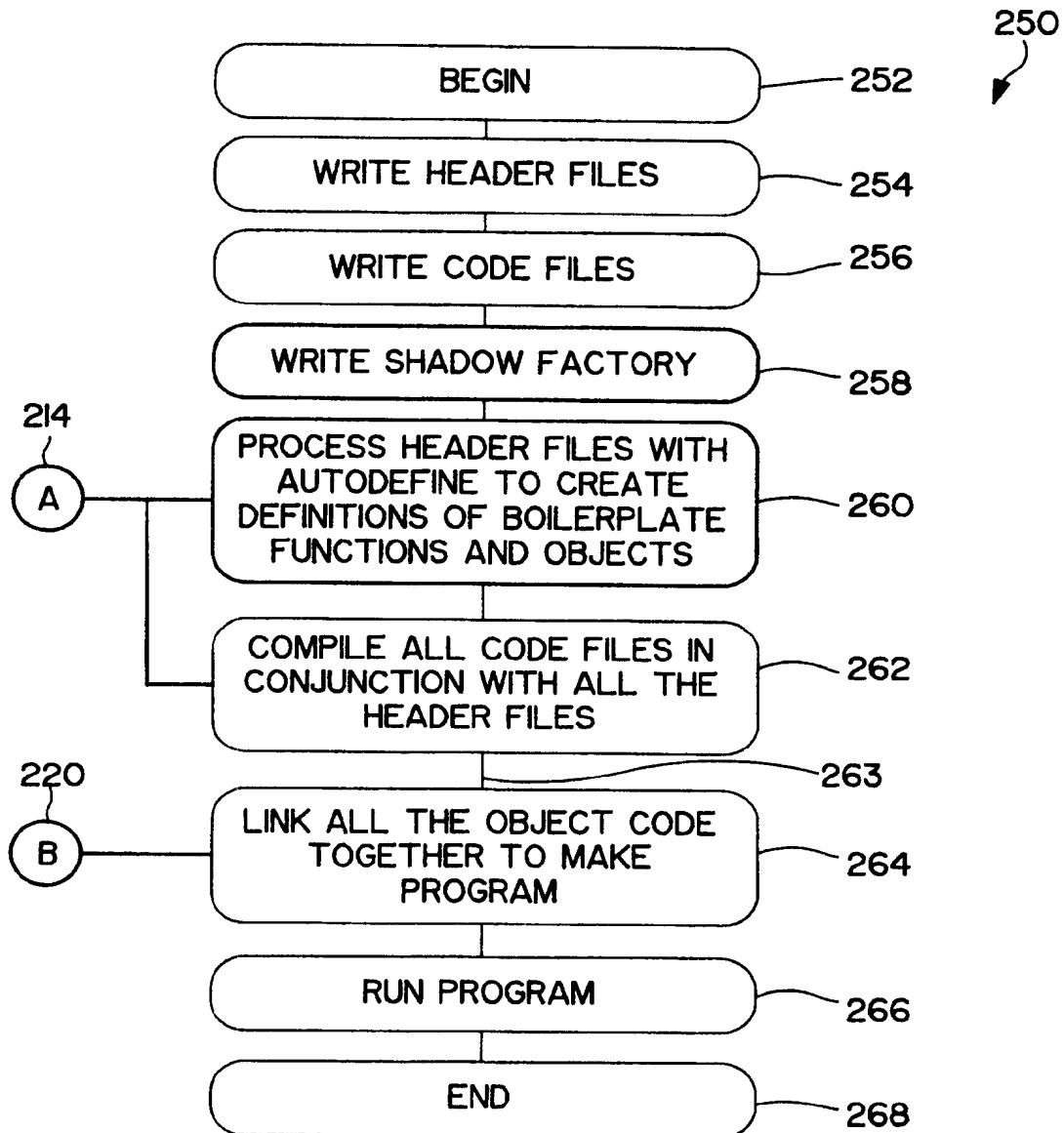


FIG. 13

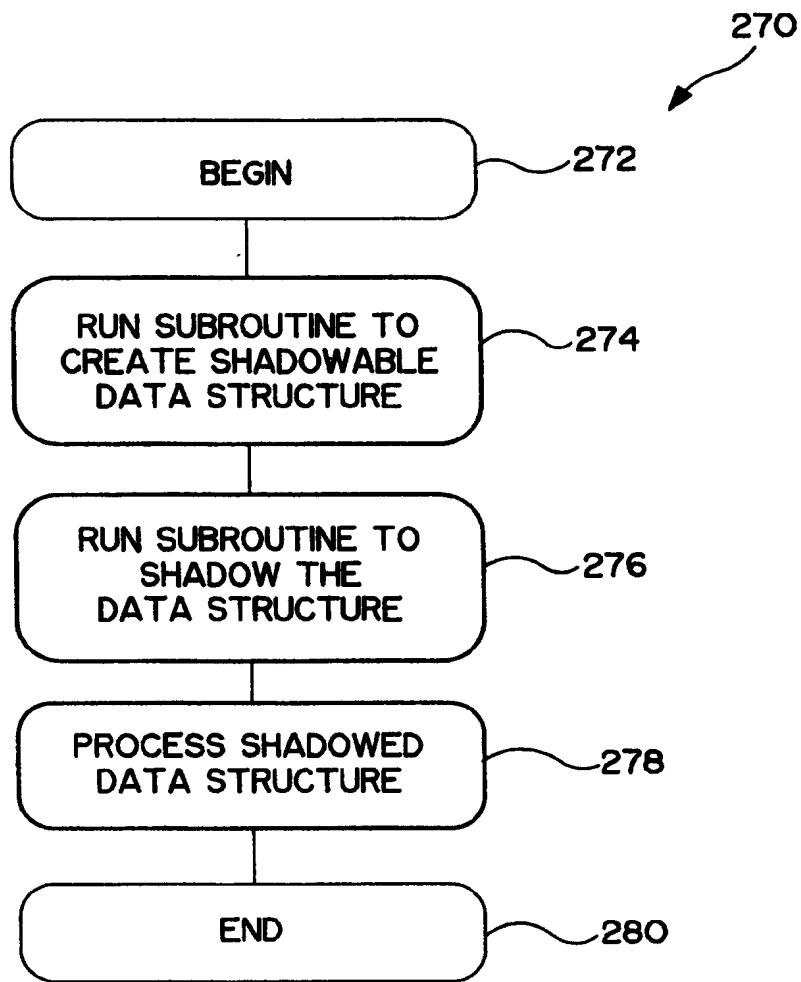


FIG. 14

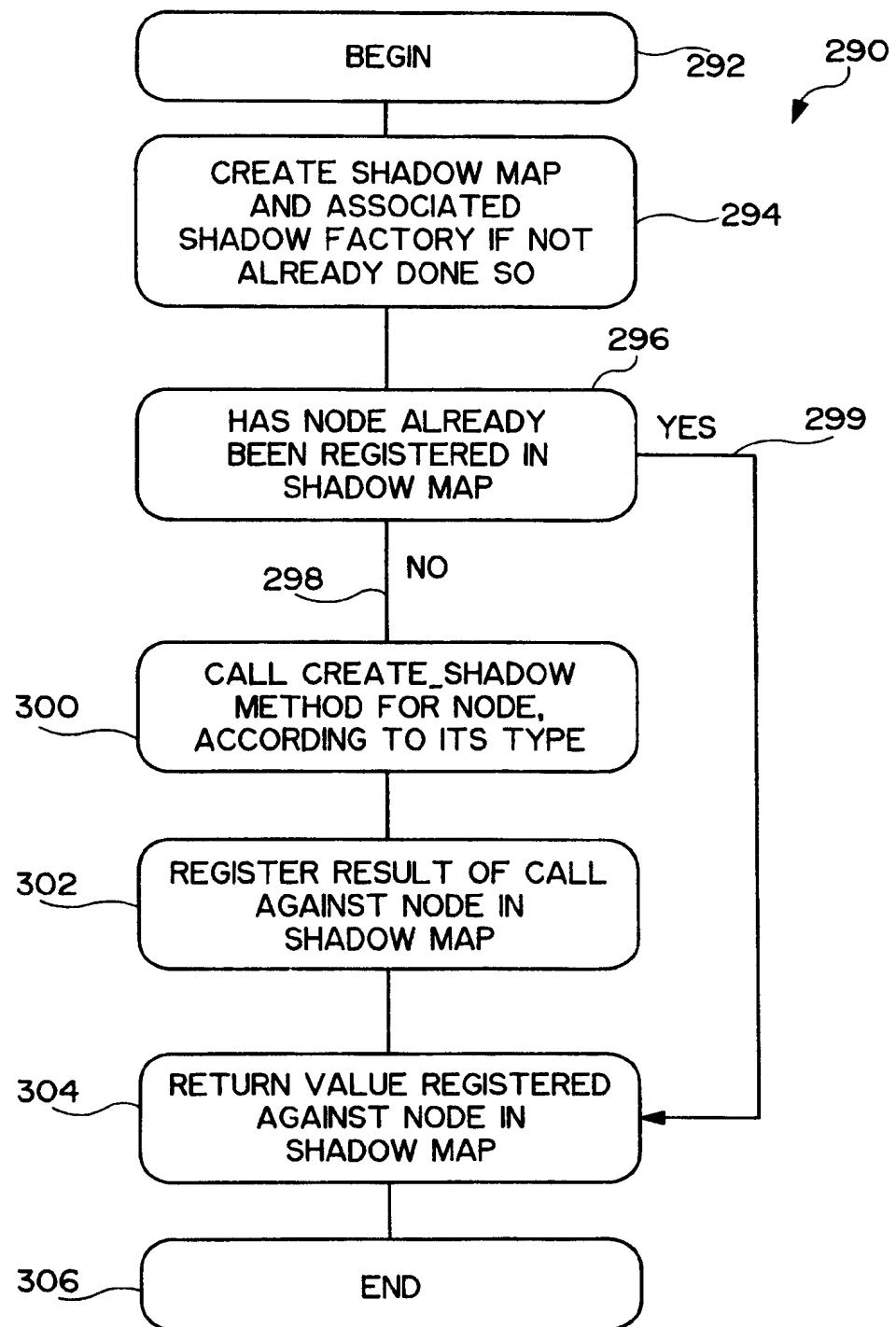


FIG. 15

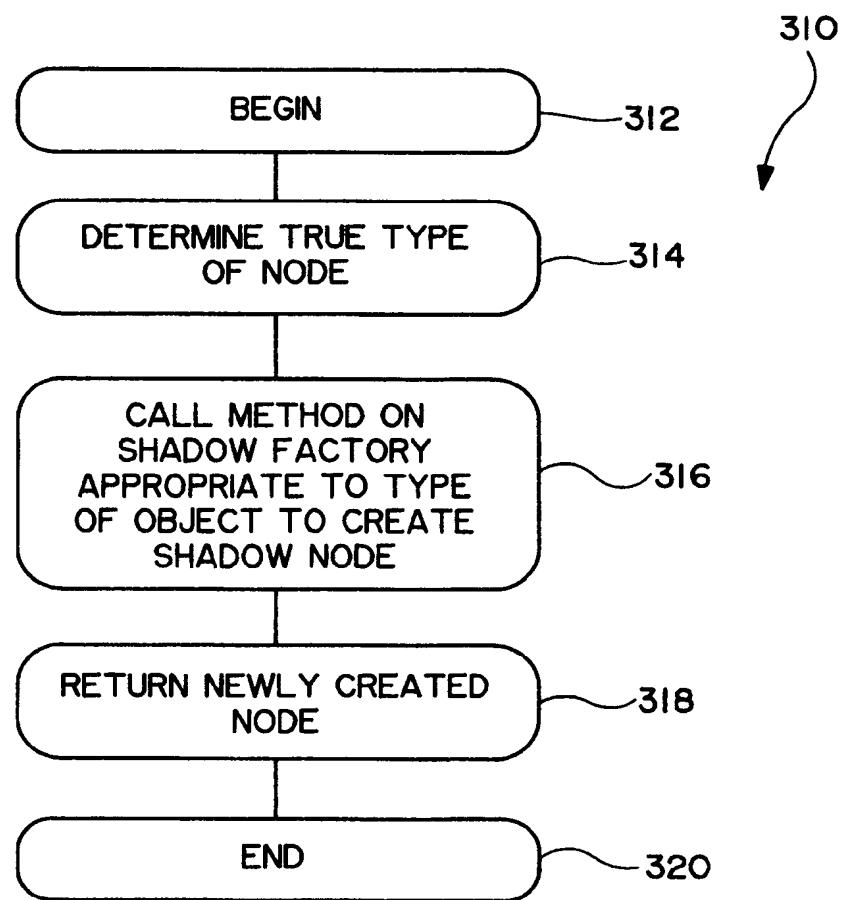


FIG. 16

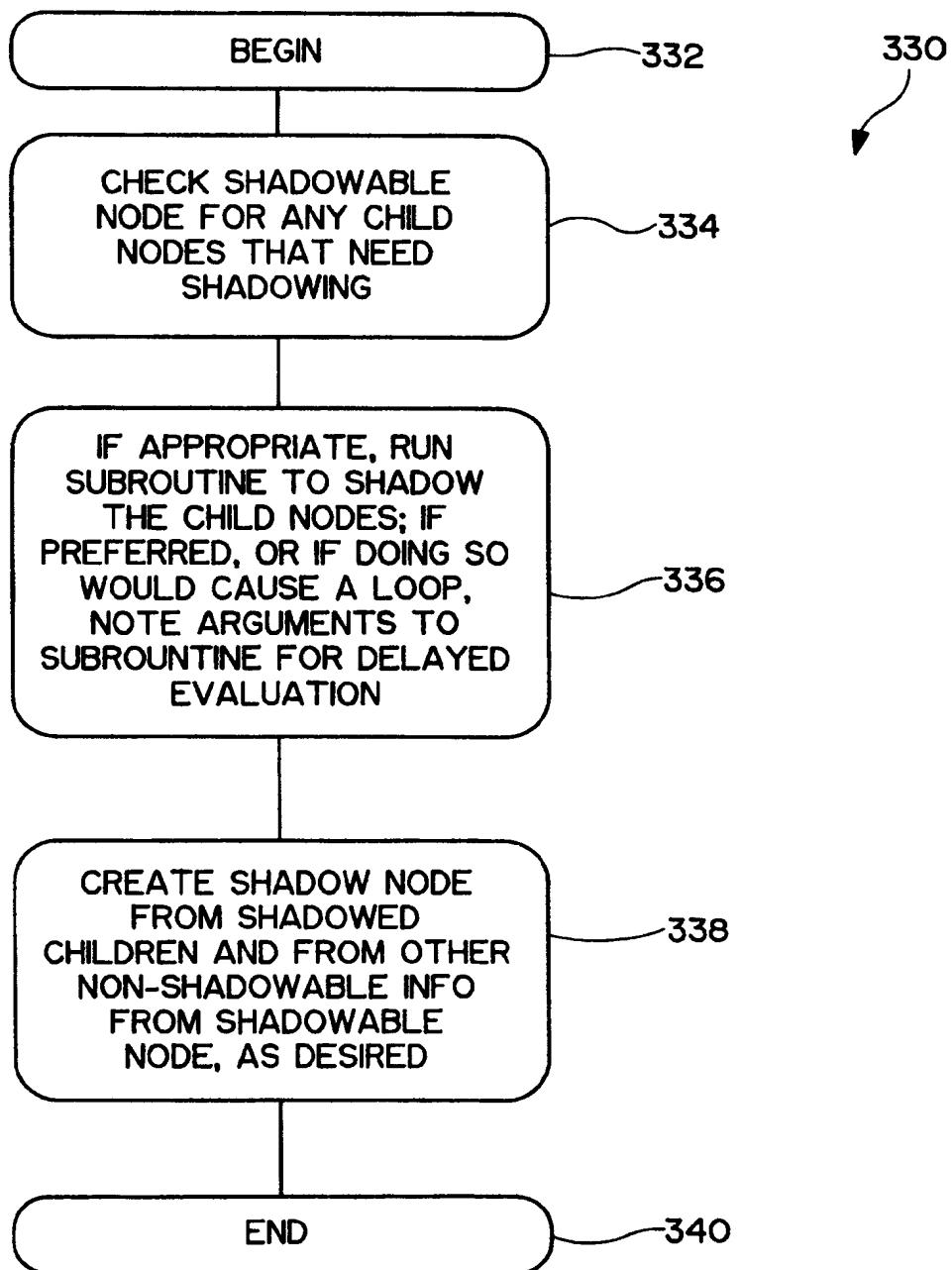


FIG. 17

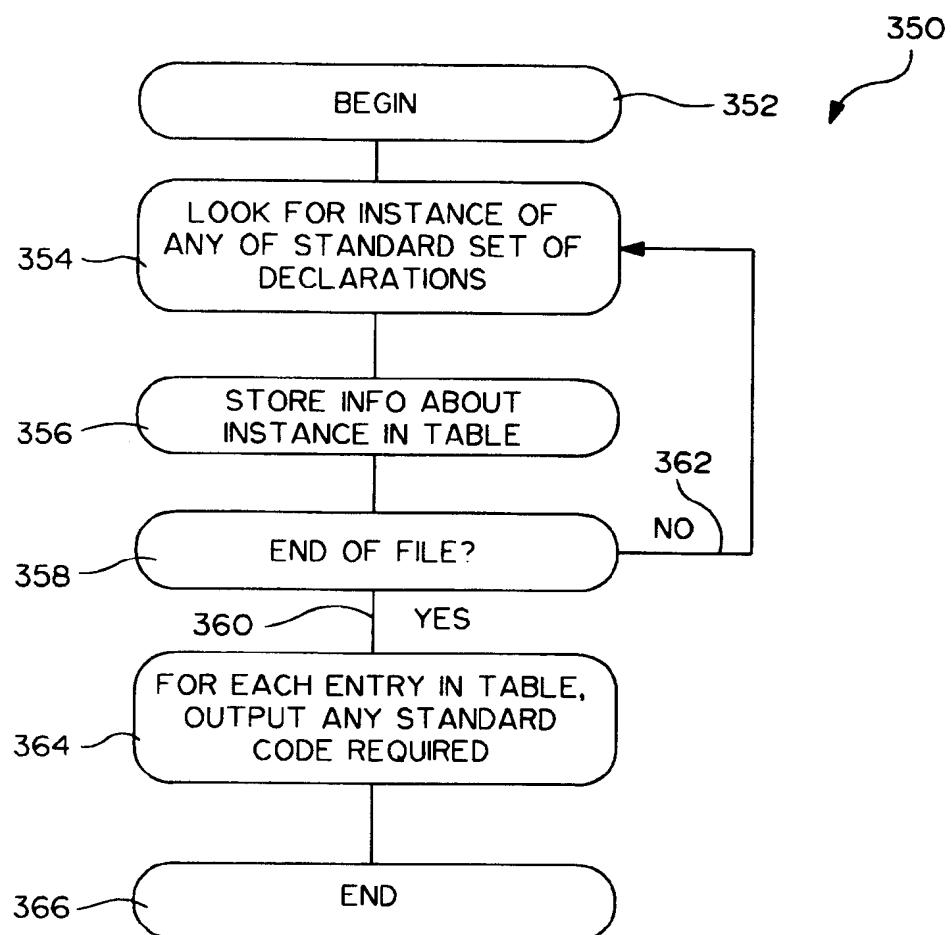


FIG. 18

<i>Class or methods</i>	<i>Responsibility</i>
LazyShadow	Standard shadow library code
Shadowable	Standard shadow library code
ShadowMap	Standard shadow library code
<i>T</i> : :create_shadow (declaration) (definition)	Data structure being shadowed Autodefine
<i>TRoot_ShadowMap</i> (class definition and default methods)	Autodefine
<i>T</i> _from_ <i>T</i> _ShadowMap (declaration and methods)	Shadow data structure
<i>T</i> : :shadow (declaration) (definition)	Shadow data structure Autodefine
<i>T</i> : :narrow (declaration) (definition)	Shadow data structure Autodefine
<i>T</i> : :narrow_to (declaration) (definition)	Shadow data structure Autodefine

Explanation of symbols

T Prototype class in shadowed data structure
TRoot Common root class for all classes in shadowed data structure
T Prototype class in shadow data structure
*T*_Factory Name of factory for shadow data structure

FIG. 19

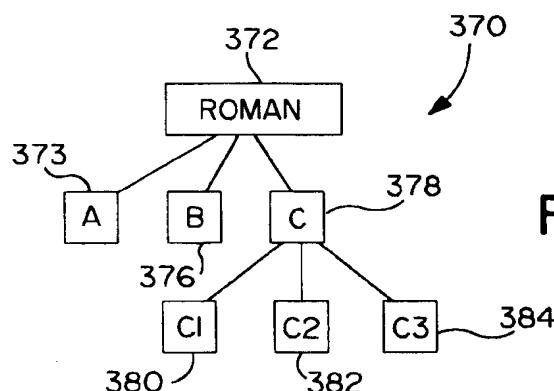


FIG. 20